

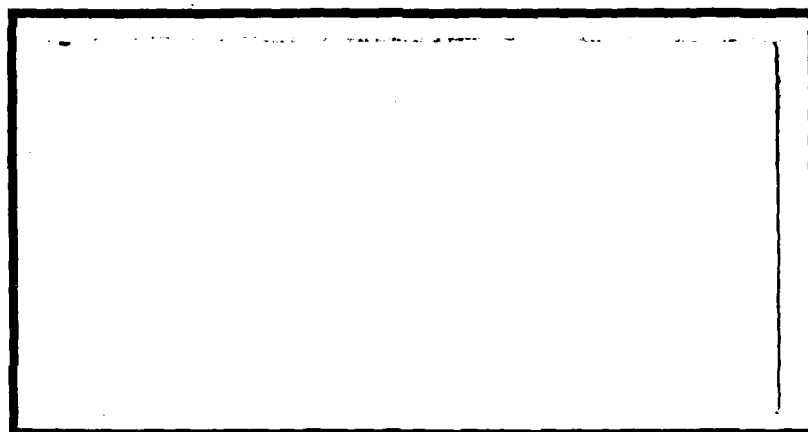
AD-A202 547

DTIC FILE COPY

1



DTIC
JAN 23 1989
S H D



DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

89 1 17 043

AFIT/GCE/ENG/88D-5

①

A DIAGNOSTIC SYSTEM BLENDING
DEEP AND SHALLOW REASONING
THESIS

James M. Skinner
Captain, USAF

AFIT/GCE/ENG/88D-5

DTIC
ELECTE
S JAN 23 1989 D
H

Approved for public release; distribution unlimited

A DIAGNOSTIC SYSTEM BLENDING DEEP AND SHALLOW REASONING

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Engineering

James M. Skinner, B.S.

Captain, USAF

DECEMBER 1988

Preface

The original objective of this thesis was to provide the engineers at the Aerospace Guidance and Metrology Center (AGMC) with an expert system capable of interfacing with the Automatic Test Equipment (ATE) used to diagnose faults in the Dual Miniature Inertial Navigation System (DMINS). In accomplishing this objective, this effort became a fascinating study into the strengths and weaknesses of two forms of diagnostic reasoning: shallow reasoning, which employs compiled knowledge to solve a problem, and deep reasoning, which solves a problem by constructing a model of the system under question. The result of this study is the Blended Diagnostic System (BDS), a system which employs both types of reasoning.

BDS was developed in four phases. In the first phase, an expert system was developed in a traditional AI manner that employs shallow reasoning. In the second phase, a model of the system was constructed and an expert system was developed that uses deep reasoning to diagnose faults based on this model. In the third phase, the two systems were blended into a system which exploits the strengths of each of the separate systems. Finally, the fourth phase added heuristics to ensure that BDS always recommends the most promising, least costly tests first, and leaves the least-promising, most costly tests as a last resort.

The success of a thesis is a measure of not only the author, but of those who supported him during the research. I would like to thank Professor F. M. Brown for his encouragement and enthusiasm that made this the most exciting project I've ever undertaken. I would also like to

		or
		<input checked="checked" type="checkbox"/>
		<input type="checkbox"/>
		<input type="checkbox"/>
Justification		
By		
Distribution/		
Availability Codes		
Dist	Avail and/or	
	Special	
A-1		

thank 1Lt Steve Rasmussen; his ideas, efforts, and support contributed greatly to this thesis. Thanks also to Dr (Capt) David Umphress for the assistance in organizing the research before it ever began and the support throughout the effort. Thanks to Capt Fautheree for reviews and technical support. Thanks also to Ken Cohen, Madeline Johnson, Jim Neri, and Wally Deskins for their help at Newark AFB, OH.

I've saved the most important thanks for last. Thanks to my children, Mallory and Jamie, who were always eager to help me "hit the keys" when I had an important deadline drawing near. And most of all, special thanks to Laura, who is my wife and best friend. Without her support and understanding, none of this would have been possible.

Table of Contents

	Page
Preface	ii
List of Figures	vi
List of Tables	viii
Abstract	ix
I. Introduction	1
Background	1
The Blended Diagnostic System	3
The Dual Miniature Inertial Navigation System	4
Methodology	4
Conclusions	7
II. Background	9
Artificial Intelligence	9
Expert Systems	9
Fault Diagnosis	11
Automatic Test Equipment	13
Applying Expert Systems to Automatic Test Equipment	14
Current Research	15
Related Theses.	18
Indications of Future Trends	19
III. The Dual Miniature Inertial Navigation System	20
Background	20
Assumptions	23
Scope	24
IV. The Shallow Reasoning System	26
Methodology	26
Identification	27
Tool Selection	28
Conceptualization	29
Knowledge Acquisition	30
Implementation	32
Testing	42
Results	42
Conclusions	43

V.	The Deep Reasoning System	44
	S.1 Deep Reasoning Scheme	44
	Knowledge Acquisition	45
	How Knowledge Acquisition Differed For The Two Types of Reasoning	49
	Program Operation	50
	Enhancements	56
	Results	58
	Conclusions	59
VI.	BDS: The Blended Diagnostic System	60
	Motive for Blending	60
	The Blended Diagnostic System	61
	Implementation	62
	Adding Heuristics to the Model	69
	Summary	75
VII.	Conclusions and Recommendations	76
	Summary	76
	Assessment	77
	Conclusions	79
	Recommendations	81
	Appendix A: Mode A Testing	83
	Appendix B: The 62 Error Messages	84
	Appendix C: The 38 Shop Replaceable Units	85
	Appendix D: Rules for XY Speed Control, YZ Speed Control, Velocity Unreasonable, and VT Greater Than 2 Knots	86
	Appendix E: The DEEP.DIAGNOSE Control Block	93
	Appendix F: Code for the Blended Diagnostic System	98
	Bibliography	186
	Vita	188

List of Figures

Figure	Page
1. Current Test Configuration	21
2. Planned Test Configuration	23
3. Decision Tree for Velocity Unreasonable	33
4. S.1 Definition of Class IMU	34
5. S.1 Definition of the Attribute ERROR.MESSAGE	35
6. The Top Level Control Block	36
7. Sample Rule from Shallow Reasoning System	36
8. A Rule to Prevent Rapid Degradation	37
9. A Consultation with S.1's Explanation Facility	39
10. Control Block to Implement Rule Categories	40
11. Basic Algorithm for Diagnosing Faults	45
12. Top Level Model of the DMINS IMU	47
13. Model of the Velocity Meter Function	47
14. Model of the Gyro Speed Control Function	48
15. Model of the Gyro Torquing Function	48
16. Top Level Control Block for Deep Reasoning	50
17. The DEEP.DIAGNOSE Control Block	51
18. Rule to Determine Bad Inputs to a Component	53
19. The BUILD.SUBCOMPONENTS Control Block	54
20. Structural Information for the Velocity Meter Function	55
21. The Attribute and Rules Required to Initiate Diagnosis from an Error Message	57

22.	Attribute and Rules Added to Improve Prompts	58
23.	Tree for Velocity Unreasonable and Vt Greater Than 2 Knots . . .	63
24.	Velocity Tree Separated into Experiential and Structural Knowledge	65
25.	Tree for XY and YZ Speed Control Errors	65
26.	Speed Control Tree Separated into Experiential and Structural Knowledge	66
27.	Rule 39 Modified to Initiate Deep Reasoning	67
28.	Modified Top Level Control Block	68
29.	Pseudo Code for the Modified DEEP.DIAGNOSE Control Block	70
30.	Rules to Order Search of Component's Inputs	74

List of Tables

Table	Page
1. Values for Risk and Test-Cost	70

Abstract

Repair of the Dual Miniature Inertial Navigation System (DMINS), a navigation system used on fast attack submarines, is the responsibility of the Aerospace Guidance and Metrology Center (AGMC) located at Newark AFB, OH. Currently, diagnostics of the system are conducted by automatic test equipment (ATE). Recent plans for upgrades of the computer that drives the ATE have made possible the integration of an expert system with the ATE, thereby increasing system reliability, decreasing test times, and improving retention of site knowledge.

The traditional approach to developing an expert system is to use shallow reasoning. Shallow reasoning, which encodes knowledge into a set of IF-THEN rules, is useful for incorporating diagnostic experience accumulated with a unit under test (UUT). It is unable, however, to accommodate situations that have not been previously encountered, and has questionable applicability to a new system, for which no experience is available.

Recent research in expert systems has emphasized the use of deep reasoning for diagnostics. A deep reasoning system diagnoses faults by constructing a model of the UUT, relying on the principle of locality and causal reasoning. Deep reasoning is suitable for a new system; if applied to an older system, however, it fails to take advantage of the lessons learned by repair-technicians.

The focus of this thesis is the development of BDS (Blended Diagnostic System), a diagnostic system which emulates a human technician

Cont

Sick

by combining shallow and deep reasoning. BDS begins by trying shallow techniques derived from the technician's experience. If these fail to determine the fault, BDS resorts to deep reasoning, constructing a model of a sub-unit of the UUT suspected to be faulty. The technician using BDS is guided around the model by heuristics based on (a) the likelihood of a component's being the cause of a failure and (b) the expected time required to test the component. This blending of shallow reasoning and heuristically-guided deep reasoning extends the class of diagnosable faults beyond the class for either form of reasoning alone, and reduces the number and duration of tests to be performed.

A prototype of BDS has been constructed for the ATE used on the Dual Miniature Inertial Navigation System (DMINS). The prototype is currently being tested at Newark Air Force Base, OH.

A DIAGNOSTIC SYSTEM BLENDING DEEP AND SHALLOW REASONING

I. Introduction

This chapter presents an overview of the research and results of this thesis effort. Each of the topics in this chapter is discussed in detail in the corresponding section of the thesis.

Background

Traditional automatic test equipment (ATE) uses a go-chain approach to testing. In this approach, the unit under test (UUT) undergoes a series of predetermined tests. At the end of each test, the results are determined. A successful test will allow the testing sequence to continue. A failure will result in the termination of the program (3:37).

Expert systems increase the effectiveness of the ATE by including information obtained from experienced diagnosticians and by providing the technician with explanations for tests (15:1). A literature search reveals that most expert systems for ATE and fault diagnosis can be placed into one of two broad categories: systems employing shallow reasoning and systems employing deep reasoning.

Shallow reasoning encodes empirical knowledge into a set of IF-THEN rules. While this is a useful method of incorporating an expert's experience with a unit under test (UUT), it is unable to accommodate situations that have not been previously encountered in existing systems, and is of questionable value in new systems where no experience is available.

A system based on deep reasoning proceeds from an understanding of the structure and function of the UUT. Common approaches to deep reasoning in fault diagnosis are reasoning from first principles, causal reasoning, and reasoning from the principle of locality (3:37). Reasoning from first principles emulates the ability of an experienced engineer or technician to troubleshoot an unfamiliar device by referring to the schematic drawings and applying appropriate physical laws. Causal reasoning entails an understanding of the mechanisms and pathways by which one component affects another in a specific type of device. The concept of locality refers to components that are connected in some manner, e.g. electrically or mechanically (5:103-104). Deep reasoning is appropriate for a new system; when applied to an older system, however, it does not take advantage of the lessons learned by technicians who may have repaired the system for some years.

Recently, attempts have been made to design systems incorporating both deep and shallow reasoning. Examples of such systems are FIS (17:68-76), IN-ATE (4:298-351), and IDM (10:188-197). FIS assists a knowledge engineer in creating a computer model of the UUT from schematics; it then allows the addition of component fault-probabilities to assist in the search for a fault. IN-ATE assists the engineer in building a model from block diagrams, then recommends the best test to conduct next based on test cost or fault-probability. IDM uses two knowledge bases, one containing shallow knowledge, the second containing deep knowledge. An executor-module determines which of the knowledge bases will work on the problem.

Another approach to the combination of deep and shallow reasoning in diagnosis is to emulate the human technician. The Blended Diagnostic System (BDS) was developed with this approach in mind.

The Blended Diagnostic System

The Blended Diagnostic System (BDS) uses both shallow and deep reasoning to emulate the way a human technician goes about diagnosing a fault. When confronted with a fault a technician will normally (1) attempt a quick fix based on his experience with such a failure in the past. If the quick fix does not solve the problem he will (2) consult the manual to determine which components could cause such a problem and then (3) test the suspect components in a order that is based on which one is most likely to be at fault.

BDS diagnoses a fault in a corresponding manner. First, BDS will (1) use shallow techniques derived from the technician's experience to imitate the technicians initial "quick fix" to repair the fault. If such repair is not successful, BDS will (2) resort to deep reasoning, constructing a model of the UUT, similar to the way a technician resorts to consulting a manual. Finally, BDS will (3) use heuristics based on the likelihood of a component's being the cause of a failure (based on MTBF) and the cost to test the component (based on time required to test the component); these heuristics guide the technician from place to place in the model. This blending of shallow reasoning and heuristically-guided deep reasoning extends the class of diagnosable faults beyond those obtainable from either form of reasoning alone, and reduces the number and duration of tests to be performed.

The Dual Miniature Inertial Navigation System

The system selected to test the performance of BDS is DMINS (Dual Miniature Inertial Navigation System). First introduced in 1974, DMINS is an inertial navigation system used on fast attack submarines, oceanographic survey ships, and aircraft carriers. Repair of DMINS is the responsibility of the Aerospace Guidance and Metrology Center (AGMC) located at Newark AFB, OH. Approximately 15 DMINS inertial measuring units (IMU) are repaired each month at AGMC. The average amount of time required to test a single IMU is more than 70 hours (20:1369-1374).

Currently, fault diagnosis of the DMINS IMU is conducted by ATE that can generate any of 62 error messages designed to aid the test technician in isolating the fault. The test technician is responsible for isolating the fault to one or more of the 38 shop repairable units (SRU) that make up the IMU. The SRU is then sent to a clean room for further diagnostics and repair.

Methodology

BDS was developed in four stages. First, an expert system using only shallow reasoning was developed in the traditional manner. Second, an expert system was developed that uses deep reasoning to construct a model of a portion of the UUT. Third, the shallow and deep reasoning systems were blended. Finally, the deep reasoning portion of the system was enhanced with heuristics to guide the technician from place to place in the model. Each of the phases of development are discussed in the remainder of this chapter.

The Shallow Reasoning System. The first phase was to develop a traditional expert system employing shallow reasoning. The structure of the system is a collection of empirical associations expressed in an IF-THEN format. This system was developed using normal knowledge engineering techniques (24:136). In particular, the knowledge required to define the system was elicited in sessions with a technician possessing 14 years' experience on the DMINS system.

Decision trees were constructed for each of the 62 possible error messages that are generated by the ATE. These clarified the technician's method of reasoning, proceeding from an initial error message to location of the SRU responsible for the fault.

The information from the decision trees was encoded in S.1, a commercial expert system shell developed by Teknowledge. The result is a system with 142 production rules that captures relevant knowledge possessed by Newark's lead technician. The system is capable of handling all 62 error messages. However, as with all shallow reasoning systems, it can only isolate faults for which it has been explicitly programmed. For this reason, emphasis was placed on maintainability to allow the system to grow as new situations are encountered.

The Deep Reasoning System. In the second phase, a system was designed that employs deep reasoning. The method was developed by Randall Davis (6:137-142) and is discussed in the S.1 User's Guide (23:3.15). The scheme uses structural diagnosis to construct a model of the UUT as needed, and diagnoses faults from this model.

Structural diagnosis is implemented in S.1 through an object oriented scheme. In this scheme, the knowledge base contains information

about each component in the structure, as well as the relationships among the components.

The approach assumes that the components are connected and that there is a flow of information from one component to another. If the output from a structure is bad, it is assumed that either one of the inputs is bad, or the component itself is faulty. If one of the inputs to the component is bad, the chain of components is followed back until the faulty component is found. If all of the inputs are good, then the component, assumed to be at fault, is diagnosed.

Diagnosis of the component begins by examining it for the existence of subcomponents. If subcomponents exist, a model of the component is generated. Components are created only when needed; efficiency is thus gained by minimizing the creation of instances. When a component is found to have no subcomponents, a bad output, and good inputs, the component is determined to be faulty.

The necessary structural information concerning the DMINS IMU was derived from the organizational level technical manual. This manual decomposes the system into ten functional categories, each of which is further decomposed into the shop replaceable units (SRU) which are of interest to the technicians at Newark (8). Of these ten functions, the three most commonly found to contain the source of the fault were chosen to be modeled for the prototype.

The resulting system can diagnose four of the 62 error messages from the ATE. This system is successful in its diagnosis; however, it always initiates a full point-by-point search for the fault without considering information that, while unrelated to the structure of the IMU, is relevant to the diagnostic process. For example, the deep reasoning

system is not concerned with what test was in progress when the error message occurred, even though this information is valuable to the technician when isolating the fault.

The Blended System. In the third phase, deep and shallow reasoning were combined to create a blended expert system that exploits the strengths of each. This system emulates a human technician: shallow reasoning is used first to try and find a "quick fix"; if this is not successful, deep reasoning techniques are employed for a more thorough (and time consuming) search.

Adding Heuristics to the Model. The fourth and final phase was to enhance the system by making the search of the model more intelligent. This was done by assigning two attributes to each of the components: RISK, which is a measure of the likelihood of the component's being at fault based on MTBF (mean time between failure), and TEST-COST, which is a measure of the difficulty to test the component, based on the time to test (in person-minutes). These attributes ensure that the most-promising, least-costly tests are performed first and that the least-promising, most-costly tests are performed only as a last resort.

Conclusions

By blending the two methods of reasoning into a single coherent system, BDS is able to exploit the strengths of each of the reasoning methods. This blending of shallow and heuristically-guided deep reasoning extends the class of diagnosable faults beyond the class for either form of reasoning alone, and reduces the number and duration of tests to be performed by the technician.

The remainder of this thesis will give the reader more details on the theory, process, and results of the Blended Diagnostic System. Chapter II is a literature review of shallow and deep reasoning, as well as a survey of expert systems developed for ATE and fault diagnosis. Chapter III describes the diagnostic process for DMINS at Newark AFB. Chapters IV, V, and VI document the methodology used to develop the shallow reasoning, deep reasoning, and blended systems respectively. Finally, Chapter VII details the results and conclusions from this research, and provides recommendations for future work.

II. Background

The purpose of this chapter is to acquaint the reader with the current state of the art in expert systems applied to automatic test equipment (ATE). This is accomplished by conducting a general review of artificial intelligence, expert systems, fault diagnosis, and ATE followed by a survey of current expert systems applied to ATE.

Artificial Intelligence

Although the field of Artificial Intelligence has been in existence for over 25 years, there is still no single uniformly accepted definition for it. Minsky is credited with the most accepted definition of artificial intelligence which is "the science of making machines do things that would require intelligence if done by men" (26:3).

In the sixties, scientists attempted to create artificial intelligence by building general purpose programs for solving broad classes of problems. These programs met with only limited success. It was not until the late seventies that the scientists involved realized an intelligent program would require high-quality, specific knowledge about a narrowly defined problem area in order to be successful. The resulting special-purpose computer programs were called expert systems (24:3-4).

Expert Systems

An expert system can be defined as "a set of computer programs which emulates human expertise by applying the techniques of logical inference

to a knowledge base" (12:1). Expert systems have become the leading practical application of artificial intelligence (12:1, 22:130).

Expert systems are an attractive alternative to conventional programming languages for a variety of reasons. These include an increase in programming productivity, ease of understanding by non-programmers, extension of human capabilities, and the ability to handle tasks that are unprogrammable in conventional languages (12:3-4).

The architecture of a rule-based expert system consists of a knowledge base, an inference engine, and a user interface. The knowledge base contains the specific knowledge about the domain. The inference engine contains the knowledge required for problem-solving. The user interface allows the user to access the system.

The knowledge engineer is responsible for extracting the rules from the expert and building the knowledge into the system. Originally, it was necessary for the knowledge engineer to develop all parts of the system. Today, a knowledge engineer may choose to use an expert system shell which will contain the knowledge representation structure, inference engine, and user interface (12:28). Several such shells are available such as ART, DUCK, KEE, KES, M.1, OPS5, S.1 and Goldworks (24:107-109). The three shells most commonly used in the Air Force are Goldworks, M.1, and S.1.

Goldworks. Goldworks is a commercial shell developed by Gold Hill Computers and is written in Gold Hill Common Lisp. Goldworks supports the use of both rules and frames. Goldworks was designed to operate on a personal computer equipped with an 80386 processor. However, Goldworks

can be operated on a Z-248 (which has a 80286 processor) by installing a card known as a Hummingboard. The Hummingboard adds an 80386 processor and 12 megabytes of memory.

M.1. M.1 is a commercial shell developed by Teknowledge for rule-based representation. M.1 employs a backward chaining control scheme and an English-like language syntax. M.1 was originally implemented in PROLOG but has been rehosted in C to increase speed of execution. M.1 operates on a IBM PC or Z-248 (24:362).

S.1. S.1 is a commercial shell also developed by Teknowledge. Factual knowledge in S.1 is represented in frames, heuristic rules, and procedural control blocks. S.1 uses backward chaining controlled by a procedural command language for its inference mechanism. S.1 has a built-in certainty handling mechanism and an explanation capability. S.1 is targeted for applications such as diagnosis, recommendation and classification. Originally S.1 was developed to be run on Xerox workstations. Teknowledge released a PC version in 1986 (12:131; 24:365).

Fault Diagnosis

In order to understand how expert systems are applied to fault diagnosis, it is helpful to consider how a human diagnoses a fault. The objective of fault diagnosis is to isolate a faulty component at either the Line Replaceable Unit (LRU) or Shop Replaceable Unit (SRU) level. When a technician performs fault diagnosis he employs two distinct forms of knowledge: deep and shallow (10:188).

Deep knowledge is based on first principles, axioms, and laws (11:30). Shallow knowledge is based on heuristics and experience. As a

novice, a technician relies heavily on deep knowledge to solve a problem. As he becomes more experienced, he compiles this deep knowledge into meaningful chunks of easily accessible knowledge; thus, what was once deep knowledge becomes shallow knowledge. As an expert, the technician will consistently use this shallow knowledge, resorting to deep knowledge only when the shallow knowledge fails to solve the problem.

Reasoning techniques can be divided into similar categories. Deep reasoning implies an understanding of the device being diagnosed. Common methods of deep reasoning used in fault diagnosis are reasoning from first principles, causal reasoning, and reasoning from the principle of locality (3:37). Reasoning from first principles involves reasoning from an understanding of the structure and function of the device in question. This is similar to the ability of an experienced engineer or technician to troubleshoot an unfamiliar device by referring to schematic drawings. Causal reasoning evolves from an understanding of the mechanisms and pathways by which one component affects another in a specific type of device. The concept of locality refers to components that are connected in some manner, e.g. electrical or mechanical (5:103-104).

Shallow reasoning uses compiled knowledge based on heuristics and experience. Experience in fault diagnosis is derived from past repairs of faulty behavior in a similar unit. Heuristics are the rules-of-thumb that an expert uses to reduce an unbounded search space to a manageable size (11:31).

Traditional expert systems employ shallow reasoning. This is accomplished by encoding empirical knowledge into a set of IF-THEN rules.

To employ deep reasoning an expert system must be capable of generating a model of the system, then consulting this model. This model can be a structural model, a functional model, or a mathematical model.

Fault diagnosis is the most widespread industrial application of expert systems (12:226). Expert systems applied to fault diagnosis systems in electronics require a relatively modest effort for implementation and have achieved some of the most rapid returns in terms of the ability to do useful work (12:170). However, expert systems are not the most common method of diagnostic assistance. Long before expert systems were considered for diagnosing faults in electronic equipment, technicians used automatic test equipment (ATE).

Automatic Test Equipment

The concept of multipurpose automatic test equipment (ATE) emerged in the mid-1950's as a result of the problems in maintenance of military electronics. These problems included complexity of the equipment to be tested, high turnover of personnel, long test times, and budgetary restrictions. ATE promised to speed testing, allow for operation by low-skill operators, and to eliminate maintenance documents (15:1).

These promises have not been completely fulfilled. Although test times decreased, the speeds expected were not achieved because of the inability of the units under test to respond at computer speeds. Reduced skill requirements were realized, but this was offset by the need for a test programmer. The stack of trouble-shooting documents grew smaller, but was soon supplemented by ATE documentation. In spite of these disappointments, the military continues to support ATE because mission

effectiveness demands fewer errors and shorter test times than is possible with manual testing methods. In addition, ATE provides the capability to support missions that would otherwise be unrealizable. Although commercial applications of ATE did not begin until the early seventies, they have increased at a much greater pace than military applications and now outnumber them (15:1-4).

Traditional ATE use a go-chain approach to testing. In this approach, the unit under test (UUT) undergoes a series of predetermined tests. At the end of each test, the results are determined. A successful test will allow the testing sequence to continue. A failure will result in the termination of the program (3:37).

This type of testing has three serious limitations. First, it can not detect multiple faults. Second, the sequence of testing can not be dynamically reordered. Third, it can only reason in the forward direction (3:37-38).

Applying Expert Systems to Automatic Test Equipment

AI is a fast-emerging technology in the ATE community (9:159). Demonstrated successes in AI applied to diagnosis and reasoning (e.g. MYCIN (24:283) and INTERNIST (24:280)) have contributed to AI's popularity in the ATE community (7:129; 13:159). The ATE community is concerned with the aspect of AI that develops computers capable of problem solving, intuitive reasoning, and learning from past experiences (1:181; 13:159). The most promising areas of AI applied to ATE include

fault isolation and test sequence optimization. Application of expert systems to ATE has the potential to improve performance by up to 30% (1:181).

Expert systems promise to overcome several limitations associated with traditional ATE. Rules can continue to execute regardless of the outcome of the previous test, thus allowing for the detection of multiple faults. Since rules have no precedence of execution, dynamic reordering of the test sequence is possible. Shells that allow for backward as well as forward chaining allow the system to reason from symptoms to faults, or to postulate a fault and determine if the symptoms support this hypothesis (3:38).

Additionally, expert systems can increase the effectiveness of ATE by incorporating the technician's experience into the automatic diagnostic process. Another advantage expert systems offer is explanation facilities that can increase the technician's confidence in the system by explaining the need for a particular test.

Current Research

The topic of expert systems for ATE and fault diagnostics of electronic equipment is currently being explored by both military and commercial researchers. Several systems exist and are in varying stages of development. A survey of current systems is presented next.

ACE. The Automated Cable Expertise (ACE) identifies trouble spots in telephone networks and recommends appropriate repair and maintenance without human intervention. When a faulty cable is detected, ACE determines the type of maintenance most likely to be effective by

consulting maintenance reports, knowledge about wire centers, and network analysis strategies. Recommendations are then stored in a database for access by the cable analyst. ACE was developed by Bell Laboratories and is implemented in OPS4 and FRANZ LISP (24:253).

ART-FUL. ART-FUL is a rule-based diagnostics system developed by RCA. ART-FUL attempts to overcome the inherent limitations of automatic test equipment by using rule-based diagnostics. ART-FUL allows for forward or backward reasoning and has the ability to detect multiple faults. ART-FUL also makes the dynamic reordering of the test sequence possible. ART-FUL is implemented in ART and runs on a Symbolics 3670 (3:37-43).

ATEX. ATEX is a diagnostic expert system designed to be embedded in an ATE. Its purpose is to assist in detecting and isolating faulty modules at the depot level. ATEX identifies and ranks faults according to their likelihood. ATEX is written in C and can run on an IBM PC or be embedded within the ATE minicomputer. ATEX was developed by Intelligent Electronics Corporation and Tel Aviv University (2:153-157).

CEPS. The B-1B Central Integrated Test System (CITS) Expert Parameter System (CEPS) combines expert system technology, conventional programming and a large data base to provide a system which assists technicians in fault diagnostics on the B1-B. The CEPS prototype was developed in S.1 by Boeing Military Airplane Company (16:209).

DART. The Diagnostic Assistance Reference Tool (DART) assists in diagnosing faults in computer hardware systems. DART uses information about the structure and expected behavior of the unit to identify design flaws in new devices. DART works by attempting to generate a proof about the cause of the malfunction using an inference procedure similar to

resolution theorem proving. DART was developed by Stanford University and is implemented in MRS (Metalevel Representation System) (24:250).

FOREST. FOREST supplements automatic test equipment by assisting in fault isolation and diagnosis. FOREST uses the engineer's rules of thumb, circuit diagrams, and general electronic trouble shooting principles. FOREST's capabilities include certainty factors and an explanation facility. FOREST is implemented in PROLOG and was developed at the University of Pennsylvania in cooperation with RCA (24:256).

IDM. The Integrated Diagnostic Model (IDM) uses two knowledge bases, one containing shallow knowledge, the second containing deep knowledge. An executor-module determines which of the knowledge bases IDM will use to work on the problem. IDM is implemented in Lisp and runs on a Symbolics 3600 Lisp machine. IDM was developed by Southwest Research Institute (10:189).

In addition to these expert systems, research is underway on expert system shells for automatic test equipment. The availability of these shells increases the number of alternatives available to the knowledge engineer charged with developing an expert system for a particular automatic test system. Three of the shells under development are FIS, LES, and IN-ATE.

FIS. The Fault Isolation Shell (FIS) assists in developing an expert system designed to aid a test technician in isolating faults in electronic equipment. FIS is used by a knowledge engineer to create a computer model of the UUT from schematics and block diagrams. FIS uses this model internally to recommend tests and analyze results. The approach FIS uses to diagnose a UUT is to follow causal rules to dynamically obtain all possible causes of the unit failure. FIS has the

ability to detect multiple faults. The methods employed by FIS make it applicable to real-time control of ATE. FIS was originally written in Franz Lisp to run on a VAX 11/780 computer. In 1986, FIS was rehosted in Lucid Common Lisp and now runs on a Sun Workstation. FIS was developed at the US Naval Research Laboratory (17:68-76).

LES. The Lockheed Expert System (LES) is a framework for building expert systems designed to guide less experienced maintenance personnel in fault diagnosis of electronic systems. LES captures the expert's knowledge in production rules and supplements this with knowledge of the structure, function, and causal relations of the UUT. LES is equipped with an explanation facility (14:414-435).

IN-ATE. The INtelligent ATE (IN-ATE) is an expert system shell designed to assist in the construction of expert systems for fault diagnosis. It uses an approach known as logic modeling in which high level block diagrams are used in troubleshooting. The system applies rules supplied by an expert diagnostician as well as those generated from an internal model to isolate faults or determine the next test. The system was developed by the Automated Reasoning Corporation and is implemented in FRANZ Lisp (4:298-351; 24:342).

Related Theses

There have been several theses in the area of expert systems and ATE previously conducted at the Air Force Institute of Technology. In 1984, Ramsey developed an expert system to troubleshoot a power supply card for the F-15. Difficulties in interfacing the automatic test equipment to the Lisp Machine prevented the system from ever being implemented (19:IX-1).

Ramsey's work was continued by Estes in 1986. Estes developed a prototype using an object-oriented language (FLAVORS) on the Symbolics Lisp machine. Estes's object was "to develop a prototype which would generate Abbreviated Test Language for All Systems (ATLAS) programs." This entailed developing a "heuristic approach to finding a near optimal ordering of the tests". Although Estes greatly narrowed the scope of his research by attempting only to show the feasibility of the approach as opposed to any implementation, his goals were not realized. Testing was never completed and no studies were conducted to determine the economic benefits of an Automatic Test Program Generator (ATPG) (9:1-4).

In a separate 1986 thesis, Wunz continued Ramsey's work with a functional approach. Wunz developed a prototype of an expert system designed to diagnose electronic circuits. The prototype, developed in KEE, was successfully tested on a power supply but had several restrictions. The list of subcomponents had to be in a specified order to enable testing to progress properly and the functional rules had to be written in Lisp (25:VI-1).

Indications of Future Trends

Current studies predict AI will become a major contributor to ATE in the near future. The increasing complexity of weapon systems and the decrease in maintenance manning levels make the application of expert systems to ATE critical for the support of weapon systems in the twenty-first century (18:377).

III. The Dual Miniature Inertial Navigation System

The system selected to test the performance of the Blended Diagnostic System is the Dual Miniature Inertial Navigation System (DMINS). This chapter discusses the details of DMINS and its test environment.

Background (20:1369-1374)

The Dual Miniature Inertial Navigation System, first introduced in 1974, is an inertial navigation system used on fast attack submarines, oceanographic survey ships, and aircraft carriers. DMINS provides continuously updated attitude (roll and pitch), heading, velocity (north and east), and position (latitude and longitude) data to the other ship subsystems. The system consists of two inertial measuring units (IMUs), two Blower Assemblies, two Electrical Equipment Mounting Bases, and a Navigation Control Console (NCC).

The IMUs contain the velocity meters and gyroscopes that measure roll, pitch, heading, and velocity. The purpose of the Blower Assembly is to cool the IMU. The Electrical Equipment Mounting Base maintains an accurate alignment between the unit and the ship's surface. The NCC provides the interface between the INS and the ships computer.

When an IMU is found to be faulty at sea, it is replaced as a whole assembly. The faulty IMU is then sent to the Aerospace Guidance and Metrology Center (AGMC) at Newark AFB for repair. On average, 15 DMINS IMUs are repaired each month at AGMC. The average amount of time required to test a single IMU is more than 70 hours.

Fault diagnosis of the DMINS IMU is conducted by automatic test equipment (ATE) driven by a test controller and test program. The current test controller, an IBM 1800 main-frame, is linked to two Navigational Control Consoles (NCCs), each of which is then linked to two IMUs. Actual operation of the IMUs is controlled by the NCC as directed by the test controller. Figure 1 shows the current test configuration.

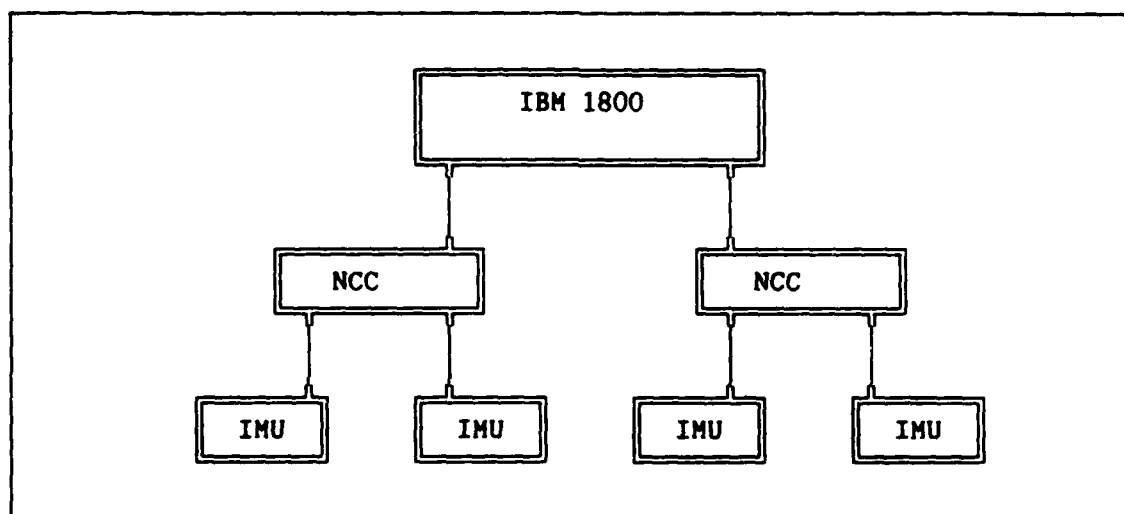


Figure 1. Current Test Configuration (20:1372).

There are three sets of diagnostic tests performed on the DMINS ATE at Newark: Mode A, a test of the system performance; Mode B, a diagnostic check to ensure that the operating parameters are within coarse limits; and Mode C, which is a test of the test equipment itself.

When an IMU arrives at AGMC, it initially undergoes Mode B testing to detect any obvious faults, such as an inoperative power supply. If the IMU passes Mode B tests, it proceeds to Mode A testing. Mode A testing consists of a series of five tests designed to determine if the IMU is

operating within specifications. Appendix A is a list and description of the Mode A tests.

If a fault occurs during testing, the ATE will print out one or more of 62 possible error messages (see Appendix B). The technician, from experience and an understanding of the system, uses these error messages to determine a course of action that will isolate the fault to one of 38 shop replaceable units (SRU) contained in the IMU. When the faulty SRU is identified, it is replaced and subsequently sent to the clean room for subcomponent diagnosis and repair. Next, the IMU is retested; if it passes, it is sent back to the field.

Due to the low number of IMUs tested each month, and the length of each test, it requires up to three years for a technician to acquire sufficient knowledge and experience to effectively diagnose system failures. As experienced engineers and technicians leave, overall site knowledge is diminished. As a result, only two technicians at AGMC are currently considered experts and are in high demand (21). AGMC has considered use of an expert system to increase the effectiveness of the technician, reverse the loss of overall site knowledge, decrease test times, and increase test reliability, but the present test configuration will not support integration between an expert system and the test controller (21).

Recently, AGMC engineers determined that the IBM 1800 computer must be replaced due to problems in maintainability and limited memory (64k). The IBM will be replaced with eight Z-248s connected together via a local area network (LAN); one Z-248 will be located at each test station, a seventh acting as an overall supervisor, and the eighth to hold a historical database (see Figure 2). This configuration increases the test

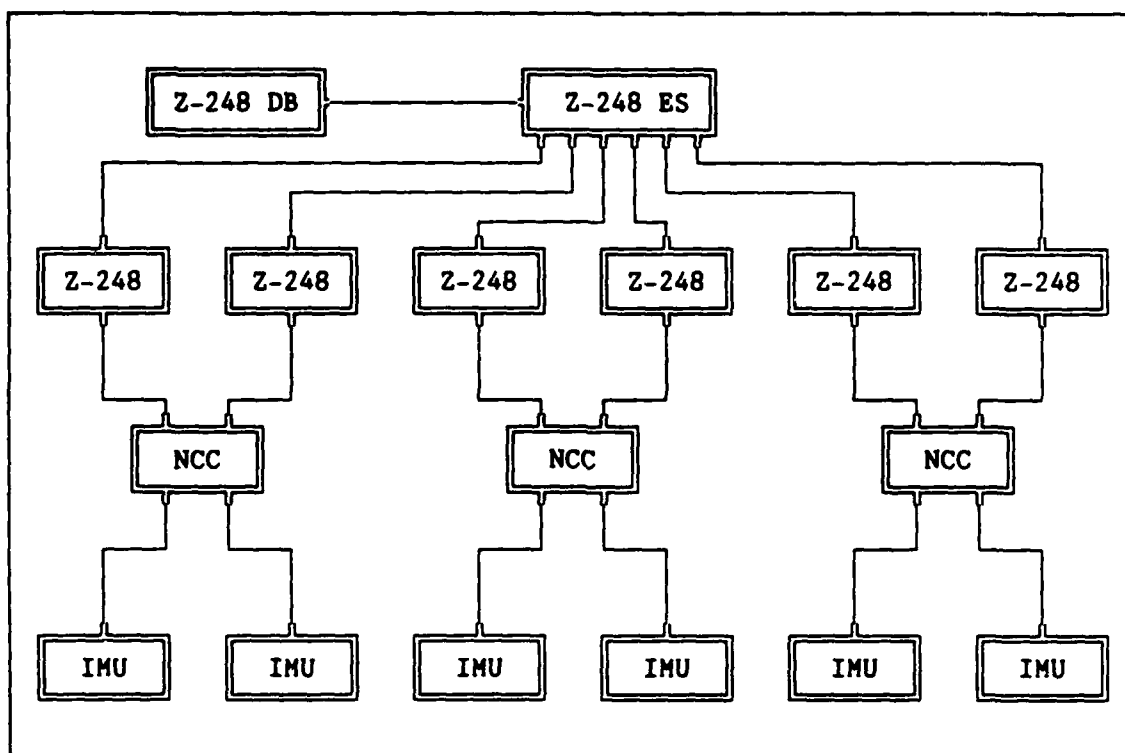


Figure 2. Planned Test Configuration (20:1373).

capability of AGMC, allowing them to test up to six IMUs simultaneously. In addition, this configuration makes it possible to integrate an expert system with the DMINS ATE. Since no expert system currently exists that is capable of being integrated with the DMINS ATE, it was a logical candidate to test the performance of the Blended Diagnostic System.

Assumptions

To ensure that the problem can be addressed with current AI tools, it is assumed that the response time of the proposed expert system is not

critical. That is, a lapse of up to ten minutes from input to output is acceptable (as implemented, the system has a lapse of no more than ten seconds).

To reduce risk of system failure due to network problems, it is assumed that all information needed by the expert system will be made available on the expert system's host computer. The expert system will not be involved with the local area network (LAN) connecting the eight computers.

The system engineer at AGMC, 1Lt Steven Rasmussen, has agreed to the foregoing assumptions.

Scope

The expert system will act as an intelligent assistant to the technician for purposes of identifying faults and suggesting further tests. The expert system developed for this thesis effort will run off-line, although future goals will be to integrate the expert system with the test controller and a historical database. Integration with the test controller will allow parameters to be passed between the test program and the expert system without involving the technician. Integration with a historical database will provide historical data to be used to identify component configuration (i.e. which gyros and velocity meters are contained in the IMU) and the cause of previous failures.

The expert system developed in this thesis effort will be limited to Mode A tests only. The sequence of Mode A tests is shown in Appendix A. Future projects could extend the expert system to include Mode B tests, or even include testing in the clean room at the subcomponent level.

Development of the local area network (LAN) and the historical database are not a part of this thesis. Both the LAN and historical database will be implemented independent of this thesis by the engineers at AGMC.

IV. The Shallow Reasoning System

This chapter documents the development of the shallow reasoning system, an expert system that encodes the experiential knowledge of the technician into a set of IF-THEN rules. The development of the shallow reasoning system is the first of four phases of development of the Blended Diagnostic System. The three remaining phases (deep reasoning, blended system, and enhancements) will be discussed in subsequent chapters.

Methodology

Before beginning development of the shallow reasoning system, an initial assessment of the problem was conducted to determine if an expert system was "possible, justified, and appropriate" (24:127). Development was shown to be possible based on the nature of the task. That is, the task is well understood and requires cognitive skills rather than common sense. In addition, experts exist, can articulate their methods, and are in general agreement. Development was shown to be justified because of the continual loss of expertise at AGMC through retirements and PCS moves. Finally, development was shown to be appropriate because of its practical value, manageable size, and heuristic nature.

Actual development of the shallow reasoning system was accomplished based on a process described by Waterman (24:136). The developmental effort was divided into six phases: identification, tool selection, conceptualization, knowledge acquisition, implementation, and testing. The results of each phase are discussed next.

Identification

The purpose of the identification phase was to specify the objectives of the expert system, and to determine the important features of the problem including the scope, required resources, and the participants in the development process (24:136).

The objective of the expert system, given any one of the error messages from Appendix B, is to assist the test technician in isolating a fault in the IMU to one of the 38 individual shop replaceable units (SRU) listed in Appendix C.

Important features of the problem were identified by interviewing the engineers and technicians at Newark AFB to determine the current method of solving the problem and to identify what aspects of the problem could best benefit from AI techniques. In addition, an extensive literature search was conducted to identify past methods of solving related problems and to identify existing expert systems used in diagnostics and automatic testing. The results of the literature search are cited in the literature review in Chapter II. The scope, presented in Chapter III, was derived from the interviews and the literature search.

Participants required were identified as a technician, a system engineer, and the knowledge engineer. The experts were identified as Wally Deskins (primary technician), Jim Neri (alternate technician), and Steve Rasmussen (system engineer). I assumed the role of knowledge engineer.

Resources required for this project were identified as a computer for development of the prototype, funds for travel, and software for

development. The computer was obtained from AFIT/EN. Funds for travel were provided by AFWAL and AFLC. Several sources offered to supply software including AFWAL (Goldworks), AFLC (S.1 and M.1), and the Navy (FIS). Several shells were thus available during tool selection.

Tool Selection

Although Waterman incorporates tool selection into the conceptualization phase, tool selection was considered important enough to merit its own phase during this project. The difficulty in selecting a tool for developing an expert system comes from a situation known as Davis's Law (24:142):

"For every tool there is a task perfectly suited to it."

The opposite is not true. For any given task there may be a number of tools that will work, or none that will work at all.

Possibilities for tools include programming languages, commercial expert system shells, or developmental research shells. For this project, it was decided to use a shell in order to take advantage of the powerful development environment associated with a shell. Shells considered included a Navy research shell (FIS), as well as three commercial shells commonly used in Air Force projects (Goldworks, M.1, and S.1). These tools are discussed in detail in the literature review in Chapter II. Of the tools considered, S.1 was selected as the development tool for the following reasons: (1) AFLC has an unrestricted license for S.1; (2) S.1 can be hosted on a Z-248 without any additional hardware; and (3) Newark AFB has an engineer already trained in S.1.

Conceptualization

During the conceptualization phase, the engineers and technicians at Newark AFB were interviewed to determine what concepts, relations, and control mechanisms are required to describe how a problem in the domain is solved.

The concepts identified in this phase included two types of diagnostic reasoning identified in the literature search: deep reasoning and shallow reasoning. Shallow reasoning, which encodes empirical knowledge into a set of IF-THEN rules, was employed in development of the system described in this chapter. Deep reasoning, which requires a knowledge of the structure of the unit under test (UUT), was used to develop the system described in Chapter V.

The relationship identified as most important during this phase was that between the error message received and the faulty component responsible for that message. This relationship offers the explanation for how a technician diagnoses a fault: an error message is generated by the ATE, the technician interprets the message and, based on experience and an understanding of the structure of the system, narrows the possibilities of what components may be the cause. He then determines a series of tests that will isolate the fault to a single component.

The most important control mechanism identified for this system is the ordering of the tests to be carried out by the technician. The order of tests affects the speed of the isolation of the fault. If components that are known to fail often are tested first, the time required to isolate the fault will normally be minimized.

Knowledge Acquisition

The purpose of the knowledge acquisition phase (referred to as the formalization phase by Waterman (24)) was to obtain the knowledge required to develop the expert system and to express this knowledge in a formal framework. For the shallow reasoning system, this entailed interviewing expert technicians in order to construct decision trees for each of the 62 error messages. These decision trees illustrate the technician's method of reasoning, proceeding from an initial error message to the location of the SRU responsible for the fault.

In preparation for the knowledge acquisition sessions, decision trees for eight common error messages were constructed. These decision trees were then encoded into an expert system prototype for the purpose of demonstrating the goal of the knowledge engineering sessions to the expert technician. Also, 62 pages were prepared for sketching the decision trees, each blank except for the name of a single error message at the top.

Three knowledge acquisition sessions were held with Jim Neri, a technician with 14 years of experience on the DMINS system. The sessions were scheduled to begin at 7 PM each evening in Mr. Neri's workplace. The location was chosen in order to keep the expert in his own domain, and in hopes of observing any details which may have seemed of little importance to the expert, such as logs, notes, or reference material. The time was chosen to allow Mr. Neri two hours from the start of his shift to attend to nightly tasks required during the shift change, in hopes of reducing interruptions.

The first session began with a demonstration of the prototype, and a discussion of our goals. The session, which lasted 3 1/2 hours, resulted in the development of decision trees for 30 of the 62 error messages. For the most part, these 30 error messages were the easiest for which decision trees could be constructed, either because they were rare error messages which required the technician to seek the supervisor's assistance early in the decision process, or because they were common messages with a simple solution. Attempts were made to construct decision trees for error messages which are difficult to diagnose, but with no success. As a result they were set aside for a later session. The limited success for this session can be attributed to my lack of experience as a interviewer, as well as Mr. Neri's lack of experience as an interviewee.

The following day, prior to Mr. Neri's arrival, the decision trees from the first session were coded, and the code was subsequently added to the existing prototype. Since time was limited, no attention was paid during coding to specifics, such as ensuring accuracy of the explanation facility. The purpose of this "rapid prototyping" was to determine the accuracy of the collected decision trees and to reinforce the goals of the second session.

The second session, which lasted about 2 hours, began with a demonstration of the prototype. Notes were taken as to how this prototype could be improved. Due to the experience gained by both parties from the previous evening, the second session was much more productive. Mr. Neri's answers were phrased in a manner more easily adapted to representation by a decision tree. The session resulted in construction of decision trees for all but three of the most difficult error messages.

On the third day the results of the second session were added to the prototype. The third session, which lasted one hour, began with a review of the prototype and notation of required changes. Mr. Neri brought with him to this session decision trees for the final three error messages. The decision trees were of such quality that they were implemented without significant change.

Figure 3 shows the decision tree for one of the 62 error messages (velocity unreasonable). The method by which the decision trees were coded into S.1 is described in the next section.

Implementation

The implementation phase, which overlapped with the acquisition phase, consisted of the actual coding of the knowledge. During this phase, the information from the decision trees was encoded in S.1. using the four basic objects in S.1: classes, attributes, control blocks, and rules.

A class is used to represent an object in the domain. Instances of a class are created during a consultation to represent individual members of the set described by the class. The code for the shallow reasoning system contains a single class called IMU for inertial measuring unit. The definition of this class is shown in Figure 4.

Attributes in S.1 are used to represent properties of classes and relationships between classes. During a consultation, the system determines values for a class's attributes. The code for the shallow reasoning system contains 117 attributes, each with a meaningful name to

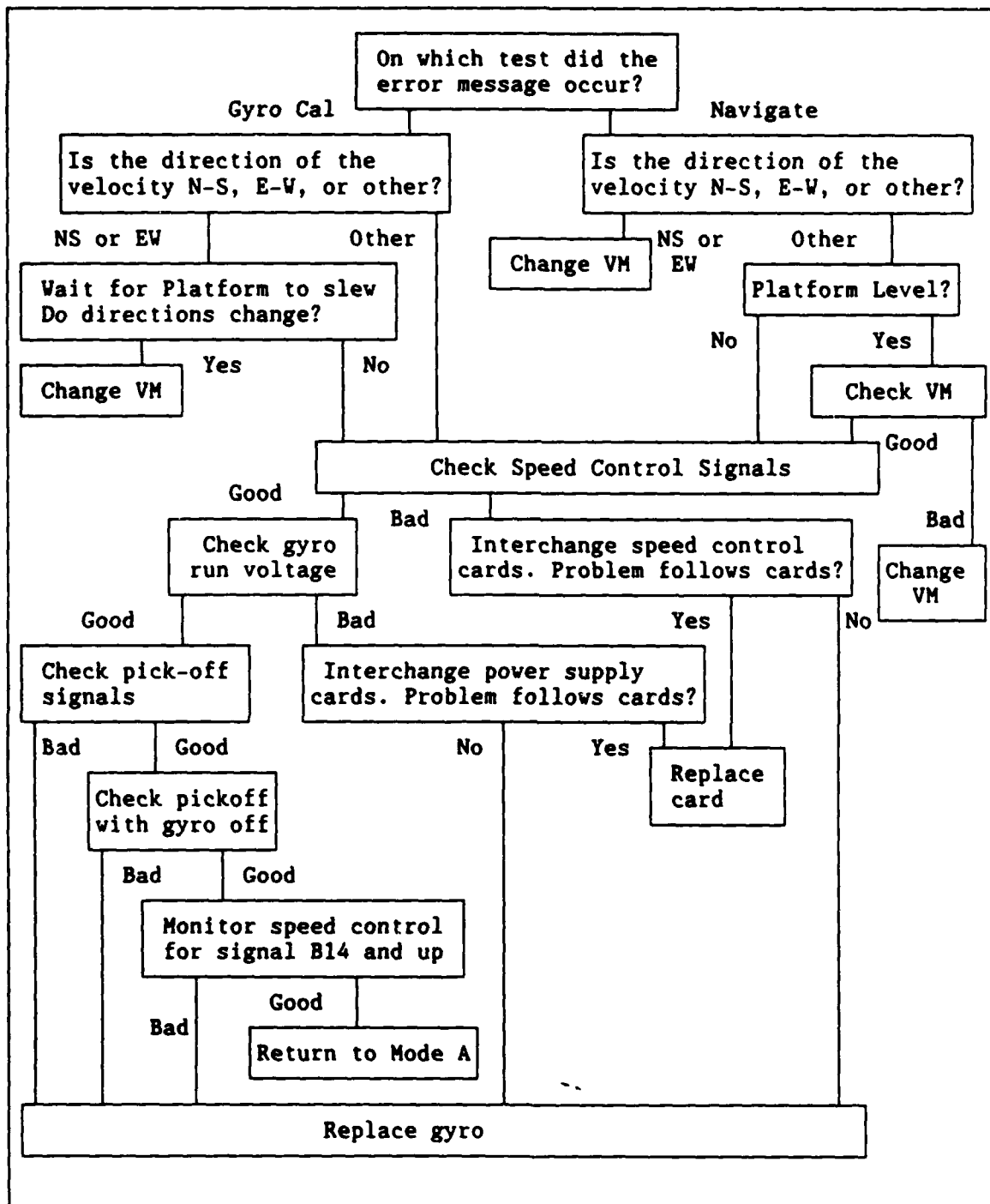


Figure 3. Decision Tree for Velocity Unreasonable.

```

DEFINE CLASS IMU
::NUMBER.INSTANCE      1
::PRINT.ID              "IMU #"
::CLASS.TRANSLATION     "an IMU "
::PLURAL.CLASS.TRANSLATION "the IMUs "
::BLAND.INSTANCE.TRANSLATION "the IMU "
::FULL.INSTANCE.TRANSLATION "this IMU"
::ANNOUNCEMENT          new.line() ! indent() ! "This IMU will
                        be referred to as: "!instance.name(IMU)
                        ! new.line() ! outdent()

END.DEFINE

```

Figure 4. S.1 Definition of Class IMU.

increase readability and maintainability. The main attributes in the code for the shallow reasoning system are ERROR.MESSAGE, ACTION, and FAULT.

ERROR.MESSAGE determines the error message generated by the ATE by querying the user. The S.1 code for this attribute is shown in Figure 5. When S.1 attempts to determine this attribute, it will do so by its legal means, {QUERY.USER}. This will cause the prompt "What error message is displayed" to appear on the screen along with a list of the legal values, i.e., the 62 error messages contained in a list called ERROR.MESSAGES. The operator can then select the appropriate message from this list.

The attribute ACTION uses rules to determine the appropriate response for the operator to take in response to an error message. FAULT is the component that the expert system determined was responsible for the error. FAULT allows an operator to query S.1 about previous faults in an IMU and provides a "hook" for integration of a historical database. This hook will make it possible to store the current fault, or retrieve the past fault of an IMU to assist in diagnosis.

DEFINE ATTRIBUTE	error.message
::DEFINED.ON	IMU
::TYPE	text
::MULTIVALUED	false
::LEGAL.VALUES	error.messages
::LEGAL.MEANS	{query.user}
::DETERMINATION.MEANS	{query.user}
::PROMPT	"What error message is displayed ?"
::TRANSLATION	"the displayed IMU Error Message"
END.DEFINE	

Figure 5. S.1 Definition of the Attribute ERROR.MESSAGE.

Control blocks in S.1 control the forward reasoning flow of the program by allowing the programmer to specify in what order the attributes should be determined. The top level control block for this system is shown in Figure 6. This control block, which is invoked automatically at the start of a consultation, creates an instance of the class IMU, then determines values for the attributes ERROR.MESSAGE and ACTION. Next it invokes an internal control block which will display the recommendations to the technician. Finally, it forces the attribute FAULT to be determined. Although the fault will normally have been concluded by this point, this statement will ensure that if no value could be found for fault, the attribute will default to NOT.DETERMINED.

Rules in S.1 are used to express heuristic or judgemental knowledge. The code for the shallow reasoning system contains 142 such rules. These rules are grouped according to the error message they are attempting to troubleshoot. This increases the maintainability of the program by ensuring that, if a new situation is encountered at Newark, required changes to the code will be localized to the area of the

```

DEFINE CONTROL.BLOCK      about.imu
::INVOCATION              top.level
::TRANSLATION             "diagnose faults in the IMU"
::BODY                    begin vars I:IMU;
                           display spaces(25) !
                           "THE DMINS FAULT ADVISOR"!
                           new.line();
                           create.instance IMU called I;
                           determine error.message[I];
                           determine action[I];
                           invoke display.recommendations(I);
                           determine fault[I];
                           end

END.DEFINE

```

Figure 6. The Top Level Control Block.

corresponding error message. Appendix D contains a listing of the set of rules that pertain to the error message velocity unreasonable. Figure 7 shows a sample rule taken from this set of rules. When this rule is fired, the technician is instructed to determine if the platform is level. If the platform is level the technician is instructed to check the velocity meter.

```

DEFINE RULE              Rule039
::APPLIED.TO             I:IMU
::PREMISE                check.for.level.platform[I] and
                           azimuth.equal.zero[I]
::CONCLUSION              check.velocity.meter[I]

```

Figure 7. Sample Rule from Shallow Reasoning System.

One disadvantage of shallow reasoning systems is that they can only diagnose situations for which they have been programmed. To prevent degradation in the event an uncoded situation is encountered, a rule was included that will detect this situation and direct the technician to see the shop supervisor. The prompt also asks that the engineering section be contacted in order to add the necessary code to handle this situation should it reoccur. This rule is shown in Figure 8.

```
DEFINE RULE rule006
::APPLIED.TO I:IMU
::PREMISE    see.shop.supervisor[I]
::CONCLUSION action[I]="this is a difficult problem to troubleshoot.
               Contact the shop supervisor for further assistance.
               Once the fault has been determined, contact the
               engineering section so this program can be updated to
               include the new information"
END.DEFINE
```

Figure 8. A Rule to Prevent Rapid Degradation.

To increase the technician's confidence in the expert system, extensive use was made of S.1's explanation facility. This allows the technician to ask why S.1 is requesting that a specific test be performed. For example, when S.1 attempts to fire rule039 (as illustrated in Figure 7) the operator is prompted to check the A to D converter. At this point the operator can ask why this is necessary. If the operator is not satisfied with the answer S.1 gives, he can request further explanation by asking why again. S.1 will backchain through the rules in an attempt to explain why it is seeking information until it reaches the

top goal of the system. Figure 9 is an excerpt from a transcript in which the operator queried S.1 for the ultimate reason for checking the A to D converter.

After coding was complete, an attempt was made to increase the efficiency of the program with rule categories. By categorizing rules, it is possible to control not only what order rules are fired in, but also whether they will be fired at all. It was reasoned that by allowing only the rules that pertained to the error message received to fire, it would speed the execution of the consultation. This extra control feature was implemented by assigning all 142 rules to one or more categories based on the error message(s) to which the rule pertained. Next, the legal means by which all of the attributes could be determined was changed to force it to use the control block called DETERMINE.ATTRIBUTE. This determination control block is shown in Figure 10. The control block restricts S.1's inference engine to consider only the rules contained in the category associated with the error message received.

The resulting program was tested and found not to decrease the execution time. It is postulated that the overhead associated with the rule categories canceled any savings obtained from the reduction in the number of rules to be searched. Because the addition of rule categories added complexity without any apparent increase in efficiency, they were removed. As the program grows, it may be worthwhile to implement rule categories.

(6) Check the A-D converter on the test station.
Is the platform level (azimuth = 0)?

m> why

I asked whether the platform is level in order to apply rule039.
Rule039 is used to determine whether check the velocity meter.

rule039:

If

check to see if the platform is level, and
it is not true that the platform is level,

then

check the velocity meter.

m> why

I need to determine whether check the velocity meter in order to
apply rule040. Rule040 is used to determine whether replace the
component determined faulty.

rule040:

If

check the velocity meter, and
the velocity meters are faulty,

then

replace the component determined faulty; and
the fault of the problem is corresponding.velocity.meter.

m> why

I need to determine whether replace the component determined faulty
order to apply rule004. Rule004 is used to determine the
recommendations for fixing the problem.

rule004:

If

replace the component determined faulty,
then

the recommendations for fixing the problem is
Recommend that the <the fault of the problem> be replaced.

m> why

I need to determine the recommendations for fixing the problem in
order to diagnose faults in the IMU .

This is the top-level goal of the system.

Figure 9. A Consultation with S.1's Explanation Facility.

```

DEFINE CONTROL.BLOCK determine.attribute
::INVOCATION determination
::ARGUMENTS a:attribute, I:IMU
::TRANSLATION "Determine the appropriate action"
::BODY      begin
              case error.message[I] of
                no.input.3.axes,
                no.input.az,
                no.input.pitch,
                no.input.roll: seek a[I] by rules
                               whose.category.is no.input;
                output.word.parity.fault,
                output.word.parity.cont,
                gyro.temp.normal,
                input.parity.fault: seek a[I] by rules
                                   whose.category.is not.fault;
                pwr.interrupt,
                automatic.shutdown: seek a[I] by rules
                                    whose.category.is power;
                xy.speed.control,
                yz.speed.control: seek a[I] by rules
                                 whose.category.is speed;
                imu.major: seek a[I] by rules
                           whose.category.is major;
                velocity.unreasonable,
                vt.greater.than.2.knots: seek a[I] by rules
                                         whose.category.is velocity;
                excess.angle: seek a[I] by rules
                              whose.category.is angle;
                servo.disable: seek a[I] by rules
                               whose.category.is servo;
                z.stab: seek a[I] by rules
                       whose.category.is stable;
                mux.decoder.dl.fault,
                cage.xy.dl.fault,
                cage.yz.dl.fault,
                gyro.start.dl.fault,
                gyro.run.dl.fault,
                uyk.good.dl.fault,
                input.parity.dl.fault.mux04,
                input.parity.no.2.dl.fault,
                output.word.parity.dl.fault.mux09: seek a[I] by rules
                                                    whose.category.is power.down;
                system.in.free.run: seek a[I] by rules
                                   whose.category.is free.run;
                imu.minor: seek a[I] by rules
                           whose.category.is other;

```

Figure 10. Control Block to Implement Rule Categories.

```

x.gyro.torque.fault,
y.gyro.torque.fault,
z.gyro.torque.fault: seek a[I] by rules
                        whose.category.is torque;
xvm.precounter.fault,
yvm.precounter.fault: seek a[I] by rules
                        whose.category.is precounter;
system.not.properly.caged: seek a[I] by rules
                        whose.category.is not.caged;

gyro.hot,
gyro.cold: seek a[I] by rules
           whose.category.is gyro.temp;
i.c.fault: seek a[I] by rules
           whose.category.is ic.fault;
rms.out.of.spec: seek a[I] by rules
                 whose.category.is out.of.spec;
plat.stab.abort: seek a[I] by rules
                 whose.category.is platform;

imu.o.load,
imu.o.temp,
dcc.o.load.o.temp,
dcc.o.temp,
comp.tie.in.sw.on,
i.c.fault.inhb.enab,
seq.cnt.no.compare,
i.c.data.loop.fault,
i.c.fault.cont,
in.parity.test.inhb.enab,
out.word.par.inhb.enab,
major.reset.fault,
minor.reset.fault,
minor.fault.cont,
both.vm.precounter.failure,
vm.bite.failure,
vt.vr.greater.than.3.knots,
minisins.vel.dif.exceeds.limit,
minisins.pos.dif.exceeds.limit,
parity.test.1.no.go,
parity.test.2.no.go,
parity.test.3.no.go,
put.intercom.test.no.go: seek a[I] by rules
                        whose.category.is rare.msg

end
end
END.DEFINE

```

Figure 10. Control Block to Implement Rule Categories (Cont'd).

Testing

The shallow reasoning expert system has a set of known entry points (the 62 error messages), a set of known end points (the 38 SRUs), and a known network that connects the entry and exit points (the decision trees). For this reason, it was possible to test the system exhaustively and to prove that it accurately reflects the information captured during the knowledge engineering sessions.

To ensure that the knowledge captured during the knowledge engineering sessions was correct, the decision trees were reviewed by Newark's other technician designated as expert, Wally Deskins. Mr. Deskins was in general agreement with the decision trees, thus no changes were deemed necessary.

The system is currently under test in the Newark shop area. Technicians compare the recommendations and conclusions of the system against actual situations and suggest modifications for improving the system. Each consultation by a technician is captured on a script for later analysis.

Results

The method of development documented in this chapter resulted in an expert system based on shallow reasoning containing 142 rules. This expert system successfully handles all 62 error messages generated by the ATE. However, as with all shallow reasoning systems, it can only isolate faults for which it has been programmed explicitly. For this reason, emphasis was placed on maintainability to allow the system to grow as new situations are encountered.

It should be noted that the quality of the resulting expert system is directly related to the availability of technicians experienced on the unit for which the expert system was built. In the case of DMINS, the required expertise was available. Had it not been available, the shallow reasoning system would have limited success.

Conclusions

Shallow reasoning encodes empirical knowledge into a set of IF-THEN rules. This is a useful method of incorporating an expert's experience with a unit under test (UUT). The quality of the resulting system, however, is dependent on the availability technicians experienced on the unit for which the system was built.

V. The Deep Reasoning System

This chapter documents the second of four phases in the development of the Blended Diagnostic System. In this phase, a system was designed that employs deep reasoning. The method was developed by Randall Davis (6:137-142) and is discussed in the S.1 User's Guide (23:3.15-3.26). The scheme uses structural diagnosis to construct a model of the UUT, refining the model as needed to locate the fault.

S.1 Deep Reasoning Scheme

As was discussed in Chapter II, a diagnostic system based on deep reasoning proceeds from an understanding of the structure and function of the UUT. The deep reasoning system discussed in this chapter performs structural diagnosis through the principle of locality. The concept of locality refers to the manner in which components are connected.

Structural diagnosis is implemented in S.1 through an object oriented scheme. In this scheme, the knowledge base contains information about each component in the structure, as well as the relationships among the components.

The approach assumes that the components are connected and that there is a flow of information from one component to another. If the output from a structure is bad, it is assumed that either one of the inputs is bad, or the component itself is faulty. If one of the inputs to the component is bad, the chain of components is followed back until the faulty component is found. If all of the inputs are good, then the component, assumed to be at fault, is diagnosed.

Diagnosis of the component begins by examining it for the existence of subcomponents. If subcomponents exist, a model of the component is generated. In this manner, components are created only when needed. This increases efficiency by minimizing the creation of instances. When a component is found to have no subcomponents, a bad output, and good inputs, the component is determined to be faulty. The basic algorithm to diagnose component(x) is shown in Figure 11.

```
If component(y) is a bad input to component(x)
then diagnose component(y);
else if component(x) has subcomponents
then build the subcomponents and
    diagnose the first subcomponent;
else component(x) is the faulty component.
```

Figure 11. Basic Algorithm for Diagnosing Faults.

Knowledge Acquisition

The objective of the knowledge acquisition phase for the deep reasoning system was to obtain the knowledge required to construct a model of the DMINS IMU. The necessary structural information concerning the DMINS IMU was available from the organizational-level technical manual (8). This manual decomposes the system into ten functional categories, each of which is further decomposed into the shop replaceable units (SRU) which are of interest to the technicians. Of these ten functions, the three most commonly found to contain the source of the fault were chosen to be modeled for the prototype. The functions chosen were the velocity

meter function, gyro speed control function, and the gyro torque function. In addition, it was necessary to consider two more functions at the top level, the timing sequence function and the platform torquing function, since the chosen functions depend on their output.

Although the organizational manual contained the necessary structural knowledge for the DMINS IMU, it was not possible to develop a model directly from this manual without a background in maintenance, as well as a deeper understanding of the components of the system, their purpose, and their relationships. For this reason, a knowledge engineering session was held with 1Lt Steve Rasmussen, the designated system engineer, to convert the schematics from the manual into a simple model.

The knowledge acquisition was accomplished in a single eight hour session (although it would have been preferable to perform the acquisition over a series of shorter sessions, time and distance constraints made this impossible). This session took place in a conference room at 1Lt Rasmussen's workplace to allow easy access to necessary reference materials.

During the knowledge acquisition session, the components of the DMINS IMU were discussed, as well as the relationships among the components. A model of the top level of the system was developed by sketching a simple block diagram on the blackboard. From this model, inputs and outputs of the chosen functions were identified. Each of these functions were subsequently modeled by identifying their subcomponents, and the manner in which they were connected. As each new component was considered, the model was updated, and existing portions were modified to reflect the relationship of existing components to the added component. The resulting top-level model of the DMINS IMU is shown in Figure 12.

Models depicting further decomposition of the velocity meter function, gyro speed control function, and gyro torquing function are shown in Figures 13, 14, and 15 respectively.

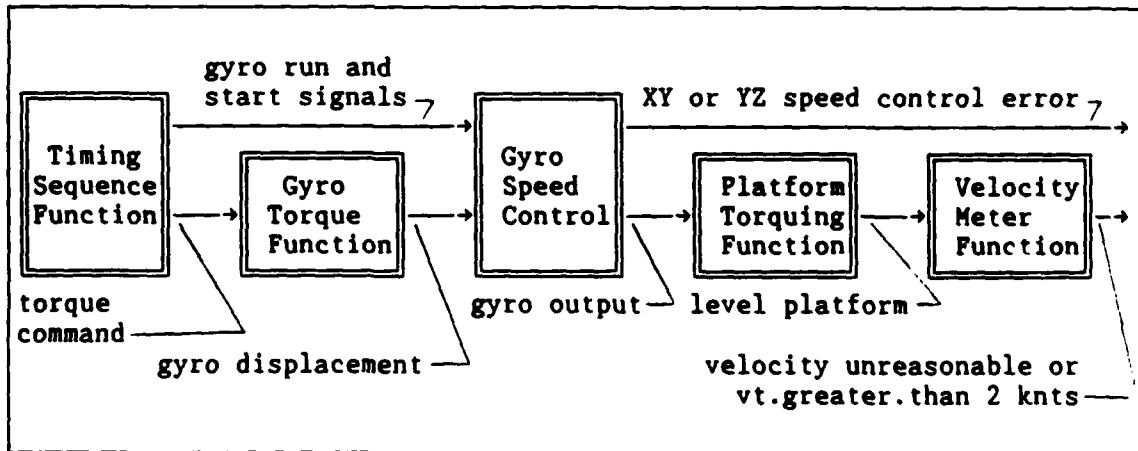


Figure 12. Top Level Model of the DMINS IMU.

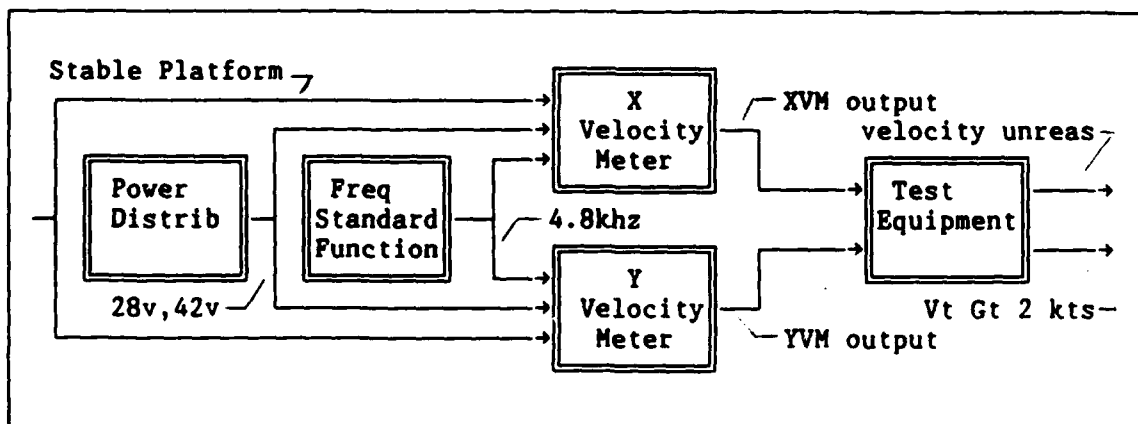


Figure 13. Model of the Velocity Meter Function.

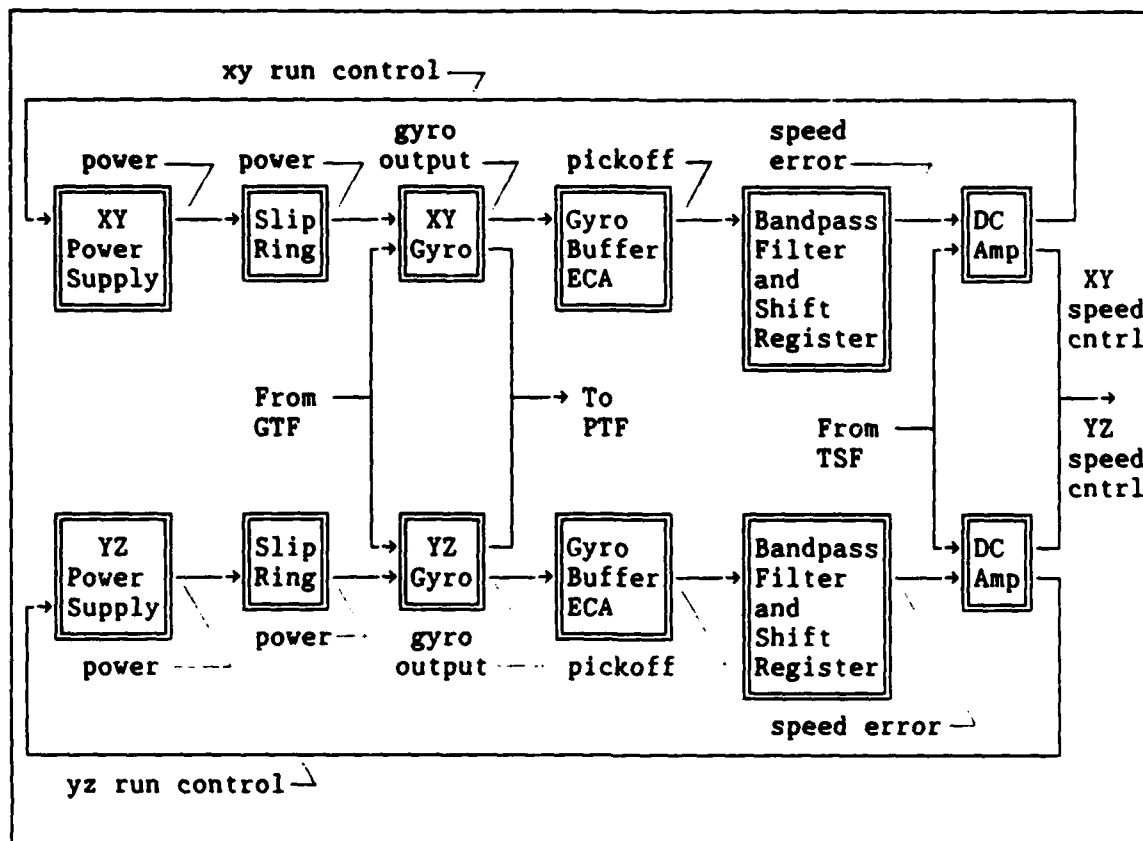


Figure 14. Model of the Gyro Speed Control Function.

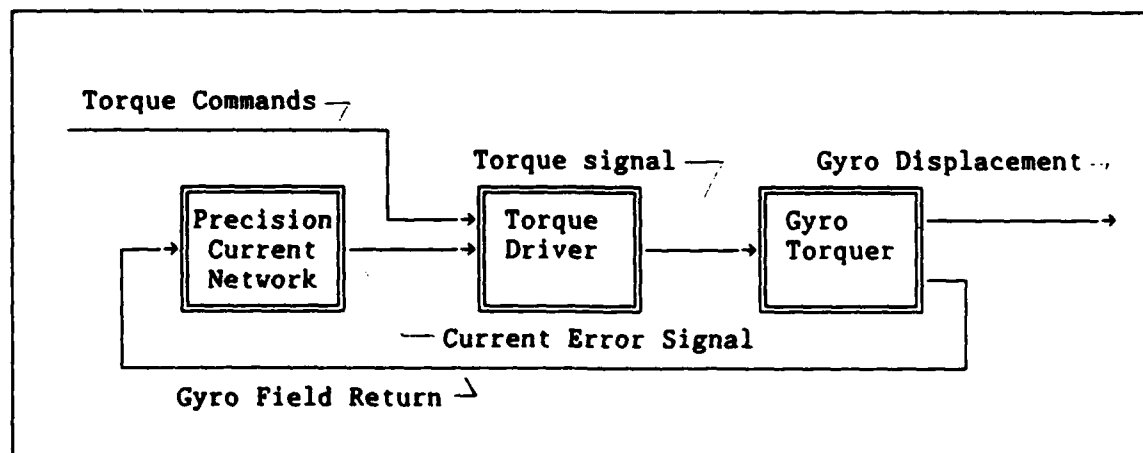


Figure 15. Model of the Gyro Torquing Function.

How Knowledge Acquisition Differed For The Two Types of Reasoning

The goal of the knowledge acquisition phase for both the deep reasoning and shallow reasoning system was to gather the knowledge required to develop the system. However, the two phases differed in that the source of the knowledge for the shallow reasoning system was an expert, whereas the source of the knowledge for the deep reasoning system was a manual, with an expert acting only as a translator.

Difficulties during the two sessions were similar. In both cases, there was a tendency for the experts to go into more detail than was necessary to build the system. Since time was constrained in both cases, it was necessary as a knowledge engineer to ensure that discussions remained relevant. This entailed a trade-off between acting as an interrogator to receive the most information in the least amount of time, risking damage to the relationship with the expert, and allowing the expert to continue, in order to let him clarify his ideas, meanwhile improving the rapport with the expert.

The technical manual was a valuable asset in the case of the deep reasoning system in that it served as a point of focus for the interview. No such focal point existed for the shallow reasoning system, except for the error message written at the top of the page of each decision tree.

In conclusion, the existence of a document as the source of knowledge in the deep reasoning system did not eliminate the need for an expert, since interpretation of the manual was necessary. Consequently, many of the problems associated with interviewing an expert remained. However, the manual provided a focal point for discussions, and was employed as a convenient return point when the discussions drifted into an unproductive state.

Program Operation

The best way to understand the program is to consider how it operates. When a consultation begins, the top level control block is invoked. This block, shown in Figure 16, creates an instance of the class IMU, assigns it a name, and declares that it has subcomponents. Next the program determines the error message by querying the user. If the error message is one of the four that corresponds to the functions in this model, the control block DEEP.DIAGNOSE, shown in Figure 17, is invoked with IMU as an argument. Otherwise, the user is notified and the program terminates.

```
DEFINE CONTROL.BLOCK          DIAGNOSE.IMU.FAULT
::INVOCATION                  top.level
::TRANSLATION                 "determine the fault in the IMU"
::BODY
  begin
    vars I:IMU;
    create.instance imu called I with
    begin
      name.of[I] = "IMU";
      has.subcomponent[I];
    end;
    determine error.message[I];
    if error.message[I] is in {velocity.unreasonable,
                               vt.greater.than.2.knots,
                               xy.speed.control,
                               yz.speed.control}
    then invoke deep.diagnose(I)
    else display new.line() ! "This error message has not been " !
                          "included in the model."
    end;
  END.DEFINE
```

Figure 16. Top Level Control Block for Deep Reasoning.

```

DEFINE CONTROL.BLOCK                DEEP.DIAGNOSE
::INVOCATION                        internal
::ARGUMENTS                        COMPONENT:imu.component
::TRANSLATION                       "diagnose component to determine"!
                                   "fault"

::BODY
begin
determine bad.input[COMPONENT];
if bad.input[COMPONENT] definite
then begin
    if exists(COMPONENT1:imu.component |
connected.to[COMPONENT1,COMPONENT] =
bad.input[COMPONENT])
then invoke deep.diagnose(COMPONENT1)
end
else begin
    if has.subcomponent[COMPONENT]
then begin
        invoke build.subcomponent(COMPONENT);
        if exists (COMPONENT2:imu.component|
same.output.as[COMPONENT,COMPONENT2])
then begin
            invoke deep.diagnose(COMPONENT2);
        end
    end
    else display new.line() ! new.line()!
        "The fault was found to be "
        ! instance.trans(component)!".";
    end
end;
END.DEFINE

```

Figure 17. The DEEP.DIAGNOSE Control Block.

The DEEP.DIAGNOSE control block implements the algorithm shown in Figure 11. It has four key attributes:

1. CONNECTED.TO[COMPONENT1,COMPONENT2] = N. This attribute indicates that the output of component1 is the Nth input of component2. This is the attribute that allows the system to chain through the network of components.

2. HAS.SUBCOMPONENT[COMPONENT]. This is a boolean attribute that is true if component can be decomposed into subcomponents. The value for this attribute is determined at the time the component is created, using information from the control blocks which describe the component.
3. BAD.INPUT[COMPONENT] = M. The value (M) of this attribute is the integer associated with the CONNECTED.TO attribute for the pair consisting of the component responsible for the bad input and the component under diagnosis. The value of BAD.INPUT is determined by trying rules.
4. SAME.OUTPUT.AS[COMPONENT1,COMPONENT2]. This attribute is true if component1 and component2 have the same output. This attribute provides the mechanism by which diagnosis continues after the subcomponents of a component under diagnosis have been constructed. Specifically, if component1 has a bad output, good inputs, and subcomponents, the subcomponents of component1 will be instantiated and diagnosis will begin with component2. At that time it will be known that component2, like component1, has a bad output.

Deep diagnosis begins by determining if the component under diagnosis has any bad inputs. This is done by firing the rule shown in Figure 18. This rule requires an additional attribute, OUTPUT.TEST.OK, which prompts the user to test the output of the component that is the input to the component under diagnosis. This is a very powerful rule. In fact, it is the only rule required to perform deep diagnosis.

DEFINE RULE	RULE.201
::APPLIED.TO	COMPONENT:imu.component
::PREMISE	already.existing(COMPONENT1:imu.component determined?(connected.to[COMPONENT1,COMPONENT]) and connected.to [COMPONENT1,COMPONENT] known and output.test.ok[COMPONENT1] thought.not)
::CONCLUSION	bad.input[COMPONENT] = connected.to[COMPONENT1,COMPONENT]
END.DEFINE	

Figure 18. Rule to Determine Bad Inputs to a Component.

All inputs to the component are tested with this rule until a bad input is found, or all of the inputs have been found to be good. If a bad input is found, DEEP.DIAGNOSE will be invoked with the component whose output is bad as the argument. This recursion will continue until a component is found with no bad inputs.

When all of the inputs to a component are found to be good, the component is checked for subcomponents. If the component has subcomponents, the program invokes the control block called BUILD.SUBCOMPONENTS, shown in Figure 19. BUILD.SUBCOMPONENTS consists of a single case statement that will invoke the control block that contains the structural information required to create a model of the component under diagnosis.

The deep reasoning program has five control blocks containing structural knowledge: one each for the IMU, Velocity Meter Function, and Gyro Torque Function, and two control blocks for the Gyro Speed Control Function. The need for the two blocks for the Gyro Speed Control Function stems from the fact that this algorithm is restricted to single-output components. Multiple-output components must be modeled by creating

```

DEFINE CONTROL.BLOCK                                BUILD.SUBCOMPONENTS
::INVOCATION                                         internal
::ARGUMENTS                                         COMPONENT:imu.component
::TRANSLATION                                       "build the logical subcomponent"!
                                                    "of the system"

::BODY
begin
  vars I:imu,
      GTF:gyro.torque.function,
      GSCFO:gyro.speed.control.function.output,
      GSCFE:gyro.speed.control.function.error,
      VMF:velocity.meter.function;

  case name.of[COMPONENT] of
    "IMU": begin
      if exists I:IMU
      then invoke build.imu(I);
      end;
    "gyro torquing": begin
      if exists GTF:gyro.torque.function
      then invoke build.gyro.torque.function(GTF);
      end;
    "gyro speed": begin
      if exists
      GSCFO:gyro.speed.control.function.output
      then invoke build.speed.control.output(GSCFO);
      end;
    "speed error": begin if exists
      GSCFE:gyro.speed.control.function.error
      then invoke build.speed.error(GSCFE);
      end;
    "velocity": begin if exists VMF:velocity.meter.function
      then invoke build.velocity.function(VMF);
      end
      end;
    end;
  end;
END.DEFINE

```

Figure 19. The BUILD.SUBCOMPONENTS Control Block.

multiple instances of the component, one for each separate output. The control block containing the structural information for the Velocity Meter Function is shown in Figure 20.

<pre> DEFINE CONTROL.BLOCK ::INVOCATION ::ARGUMENTS ::TRANSLATION ::BODY begin vars T.E:test.equipment, X.VM:x.velocity.meter, Y.VM:y.velocity.meter, P:power.distribution, F:frequency.standard; display new.line()! new.line()! "The fault has been isolated"! " to the velocity meter function. The subcomponents of the"! " velocity meter function will be constructed."; create.instance test.equipment called T.E with same.output.as[VMF,T.E]; if exists PTF:platform.torquing.function then create.instance x.velocity.meter called X.VM with begin connected.to[X.VM,T.E] = 1; connected.to[PTF,X.VM] = 1; end; if exists PTF:platform.torquing.function then create.instance y.velocity.meter called Y.VM with begin connected.to[Y.VM,T.E] = 2; connected.to[PTF,Y.VM] = 1; end; create.instance frequency.standard called F with begin connected.to[F,X.VM] = 2; connected.to[F,Y.VM] = 2; end; create.instance power.distribution called P with begin connected.to[P,X.VM] = 3; connected.to[P,Y.VM] = 3; end; end; END.DEFINE </pre>	<pre> BUILD.VELOCITY.FUNCTION internal VMF:velocity.meter.function "build the logical subcomponent"! "of the velocity meter function" </pre>
--	--

Figure 20. Structural Information for the Velocity Meter Function.

After the subcomponents of a component have been built, DEEP.DIAGNOSE is invoked with the argument set equal to the component which has the same output as the parent component. This recursive diagnosis continues until a component is found that has a bad output, no bad inputs and no subcomponents. At this point it is reasoned that this component is the fault in the system.

Enhancements

As stated earlier, the rule shown in Figure 18 is the only rule required to make the program run. However, rules and attributes were added to tailor the algorithm to DMINS. For example, the attribute ERROR.MESSAGE and rules 202 through 205, shown in Figure 21, were added to provide entry into the program directly from the error message. When the error message is determined, the program will automatically mark the output of the functions as good or bad, depending on the relationship between the error message and the function.

Additional rules and attributes were included to increase the readability of the prompts. For example, the addition of rule207, rule208, and the attribute LEVEL.PLATFORM, shown in Figure 22, changed the prompt from "Is the output of the platform torquing function good?" to "Is the platform level?". The latter prompt is much more meaningful to the technician.

```

DEFINE ATTRIBUTE                                ERROR.MESSAGE
::DEFINED.ON                                   I:IMU
::TYPE                                          text
::MULTIVALUED                                  false
::LEGAL.VALUES                                 error.messages
::LEGAL.MEANS                                  {query.user}
::DETERMINATION.MEANS                         {query.user}
::PROMPT                                       "What error message is displayed?"
::TRANSLATION                                  "the displayed IMU Error Message"
END.DEFINE

DEFINE RULE                                     RULE.202
::APPLIED.TO                                  velocity.meter.function
::PREMISE                                     already.existing(I:imu|
error.message[I] is in {velocity.unreasonable,
                        vt.greater.than.2.knots})
::CONCLUSION                                  output.test.ok[velocity.meter.function]<-1.0>
END.DEFINE

DEFINE RULE                                     RULE.203
::APPLIED.TO                                  gyro.speed.control.function.error
::PREMISE                                     already.existing(I:imu|
error.message[I] is in {velocity.unreasonable,
                        vt.greater.than.2.knots})
::CONCLUSION                                  output.test.ok[gyro.speed.control.function.error]
END.DEFINE

DEFINE RULE                                     RULE.204
::APPLIED.TO                                  velocity.meter.function
::PREMISE                                     already.existing(I:imu|
error.message[I] is in {xy.speed.control,
                        yz.speed.control})
::CONCLUSION                                  output.test.ok[velocity.meter.function]
END.DEFINE

DEFINE RULE                                     RULE.205
::APPLIED.TO                                  gyro.speed.control.function.error
::PREMISE                                     already.existing(I:imu|
error.message[I] is in {xy.speed.control,
                        yz.speed.control})
::CONCLUSION                                  output.test.ok[gyro.speed.control.function.error]<-1.0>
END.DEFINE

```

Figure 21. The Attribute and Rules Required to Initiate Diagnosis from an Error Message.

```

DEFINE ATTRIBUTE          LEVEL.PLATFORM
::DEFINED.ON             platform.torquing.function
::TYPE                   boolean
::MULTIVALUED            false
::LEGAL.MEANS             {query.user}
::DETERMINATION.MEANS     {query.user}
::PROMPT                  "Is the platform level?"
::TRANSLATION             "the platform is level"
END.DEFINE

DEFINE RULE               RULE.207
::APPLIED.TO             platform.torquing.function
::PREMISE                 level.platform[platform.torquing.function]
::CONCLUSION              output.test.ok[platform.torquing.function]
END.DEFINE

DEFINE RULE               RULE.208
::APPLIED.TO             platform.torquing.function
::PREMISE                 not level.platform[platform.torquing.function]
::CONCLUSION              output.test.ok[platform.torquing.function]<-1.0>
END.DEFINE

```

Figure 22. Attribute and Rules Added to Improve Prompts.

Results

The resulting system can diagnose the following four of the 62 error messages from the ATE: xy speed control, yz speed control, velocity unreasonable, and vt greater than 2 knots. This system is successful in its diagnosis; however, it always initiates a full point-by-point search for the fault without considering information that, while unrelated to the structure of the IMU, is relevant to the diagnostic process. For example, the deep-reasoning system does not concern itself with what test was in progress when the error message occurred, even though the decision tree in Figure 3 from Chapter IV shows that this information is valuable to the technician.

Another point to note is that the system makes no use of information based on the technician's experience. For example, based on the heuristics from the shallow reasoning system, a YZ speed control error message, unlike an XY speed control error message, is not an indication of a fault unless it has occurred more than twice. The deep reasoning system treats the two error messages identically, initiating a full point to point search, perhaps unnecessarily.

Finally, the pure model-based approach uses only the principle of locality in generating a path to search for the system fault. No consideration is given to the component's likelihood of failure, or to the difficulty of testing the component. A slip ring may be tested before a gyro even though, based on their mean time between failures (MTBF), the gyro is more than seven times as likely to be at fault, but can be tested in a twelfth of the time (21).

Conclusions

A deep reasoning system is capable of diagnosing the fault in a unit by employing the principle of locality.

Deep reasoning does not take advantage of the lessons learned by technicians who may have repaired the system for some years.

Deep reasoning does not consider information that, while unrelated to the structure of the UUT, is valuable in isolating the fault.

When ordering the tests, pure model-based search does not take into consideration the likelihood of a component's being at fault, nor does it consider how difficult testing a component may be.

VI. BDS: The Blended Diagnostic System

This Chapter documents the final phases of development of BDS. In the third phase, the systems from Chapter IV and V were blended into a single system. Phase 4 enhanced this system by adding heuristics to guide the technician in an intelligent search for the fault.

Motive For Blending

Chapter IV documented the development of an expert system for the DMINS ATE following the traditional method described by Waterman (24). This method resulted in a system based on shallow reasoning that contained 142 rules. While this system successfully handles all error messages generated by the ATE, it can only diagnose situations for which it has been explicitly programmed.

Chapter V documented the development of an expert system for the same ATE using pure deep reasoning techniques. This method resulted in the development of an expert system that constructs a model of the Inertial Measurement Unit to reason about possible faults. Although the deep reasoning system is capable of reasoning in situations for which it was not explicitly programmed, it is limited in that: (1) it does not take advantage of the lessons learned by experienced technicians; (2) it neglects information that, while not related to structure, is nonetheless useful for isolating the fault of the system; and (3) when ordering the tests, it does not consider the likelihood of a component's being at fault, nor does it consider how difficult testing a component may be.

Recently, attempts have been made to design systems incorporating both deep and shallow reasoning. Examples of such systems are FIS (17:68-76), IN-ATE (4:298-351), and IDM (10:188-197). FIS assists a knowledge engineer in creating a computer model of the UUT from schematics; FIS then allows the addition of component fault-probabilities to assist in the search for a fault. IN-ATE assists the engineer in building a model from block diagrams, then recommends the best test to conduct next based on test-cost or fault-probability. IDM uses two knowledge bases, one containing shallow knowledge, the second containing deep knowledge. An executor module determines which of the knowledge bases will work on the problem.

Another approach to the combination of deep and shallow reasoning in diagnosis is to emulate the human technician. The Blended Diagnostic System (BDS), was developed with this approach in mind.

The Blended Diagnostic System

The Blended Diagnostic System (BDS) uses both shallow and deep reasoning to emulate the way a human technician goes about diagnosing a fault. When confronted with a fault a technician will normally (1) attempt a quick fix based on his experience with such a failure in the past. If the quick fix does not solve the problem he will (2) consult the manual to determine which components could cause such a problem and then (3) test the suspect components in an order that is based on which one is most likely to be at fault.

BDS diagnoses a fault in a corresponding manner. First, BDS (1) uses shallow techniques derived from the technician's experience to imitate the

technician's initial "quick fix" to repair the fault. If such repair is not successful, BDS (2) resorts to deep reasoning, constructing a model of the UUT, similar to the way a technician resorts to consulting a manual. Finally, BDS (3) uses heuristics based on the likelihood of a component's being the cause of a failure (based on MTBF) and the cost to test the component (based on time required to test the component) to guide the technician from place to place in the model. This blending of shallow and heuristically-guided deep reasoning extends the class of diagnosable faults beyond those obtainable from either form of reasoning alone, and reduces the number and duration of tests to be performed. Appendix F lists the code for BDS.

Implementation

As was shown in Chapter IV, the first phase developed a shallow reasoning system by encoding empirical knowledge gathered from the technicians during knowledge engineering sessions. During these sessions the technician's method of diagnosis, working from each error message to a shop replaceable unit (SRU), was captured in the form of decision trees. Four of these error messages are common to both the deep and shallow systems developed in this thesis: velocity unreasonable, vt greater than 2 knots, XY speed control, and YZ speed control.

The decision tree for velocity unreasonable, first shown in Figure 3 of Chapter IV, is presented again in Figure 23 for the convenience of the reader. This tree also applies to the error message vt greater than 2 knots. A close examination reveals that the knowledge contained within the

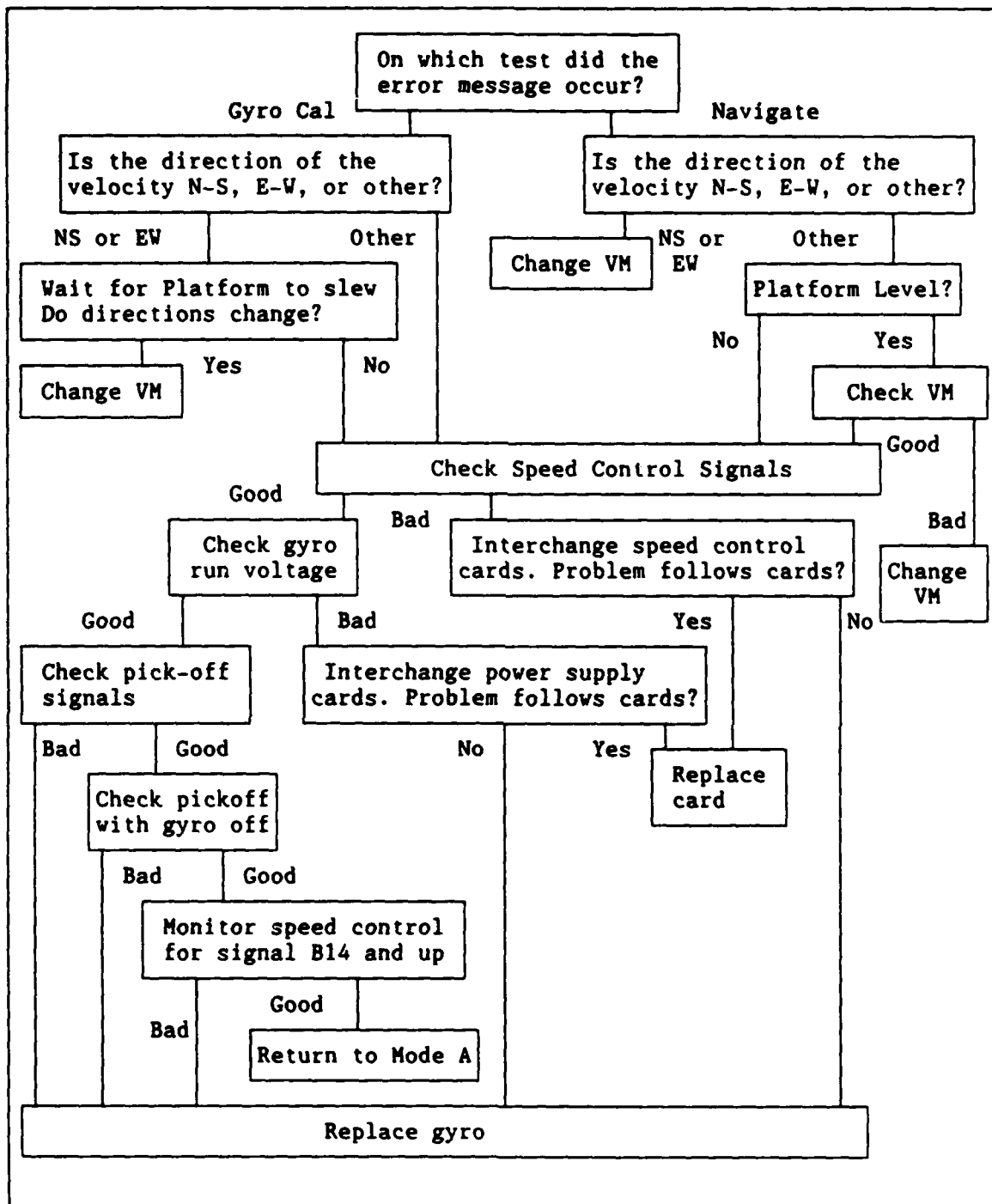


Figure 23. Tree for Velocity Unreasonable and Vt Greater Than 2 Knots.

decision tree can be separated into (a) knowledge derived from experience that results in "quick fix" tactics and (b) knowledge based on structural relations that require testing.

As an example of knowledge arising from experience, consider the prompt relating to the current test. This prompt does not pertain to structure, however, the technician knows from experience that this information is useful in isolating the fault. As an example of quick fix tactics note that no test is required to determine the direction of the velocity; the direction can be determined by reading an indicator. Likewise, determining if the platform is level, or waiting for the platform to slew requires no testing.

This portion of the knowledge was classified as experiential, or shallow knowledge. Note that the remainder of the decision tree in Figure 23 deals with structural knowledge. Figure 24 shows the line dividing the experiential from the structural knowledge in the decision tree.

The decision tree for the error messages XY speed control, and YZ speed control is shown in Figure 25. This tree is similar to the tree in Figure 23 in that the knowledge contained can be separated into experiential and structural knowledge. From experience, the technician knows that the YZ error message is not indicative of a fault unless it occurs more than twice. This information was not available to the deep reasoning system; in fact, the deep reasoning system will attempt to isolate a fault regardless of the number of previous occurrences. The remainder of the tree has information concerning structural knowledge. Figure 26 shows this tree divided into experiential and structural knowledge.

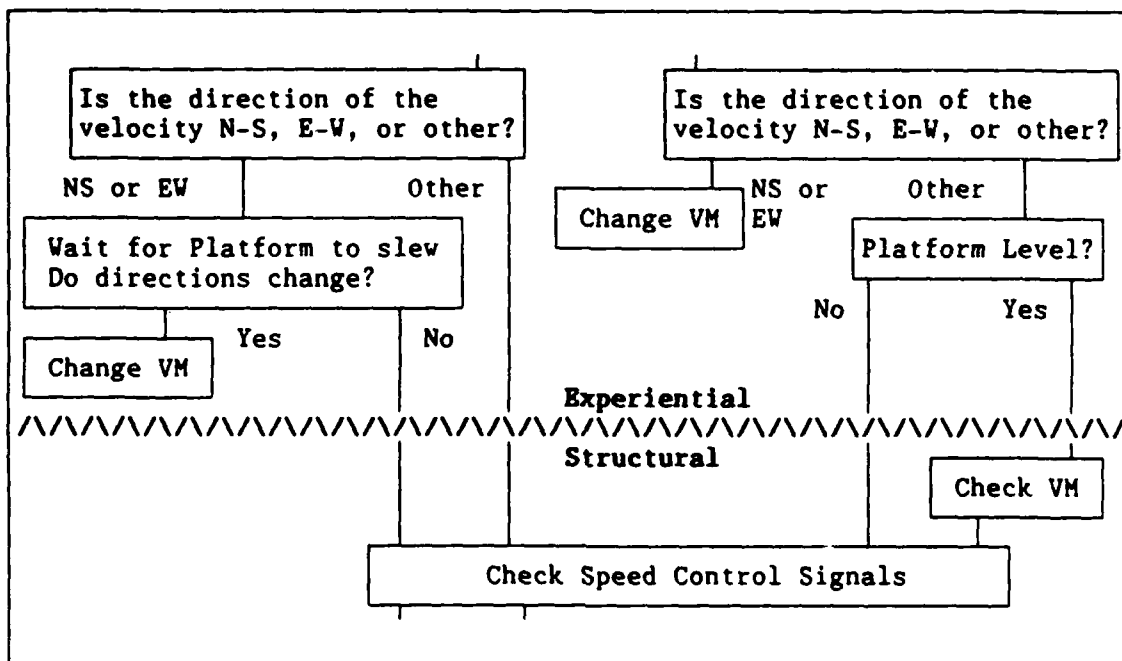


Figure 24. Velocity Tree Separated Into Experiential and Structural Knowledge.

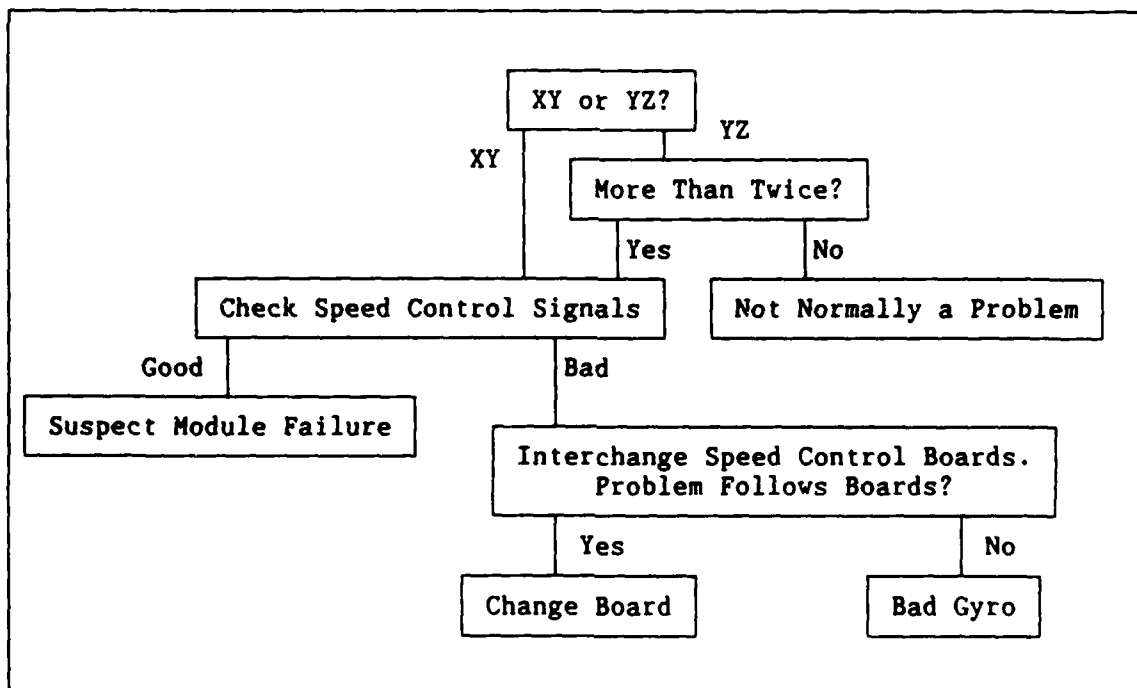


Figure 25. Tree For XY and YZ Speed Control Errors.

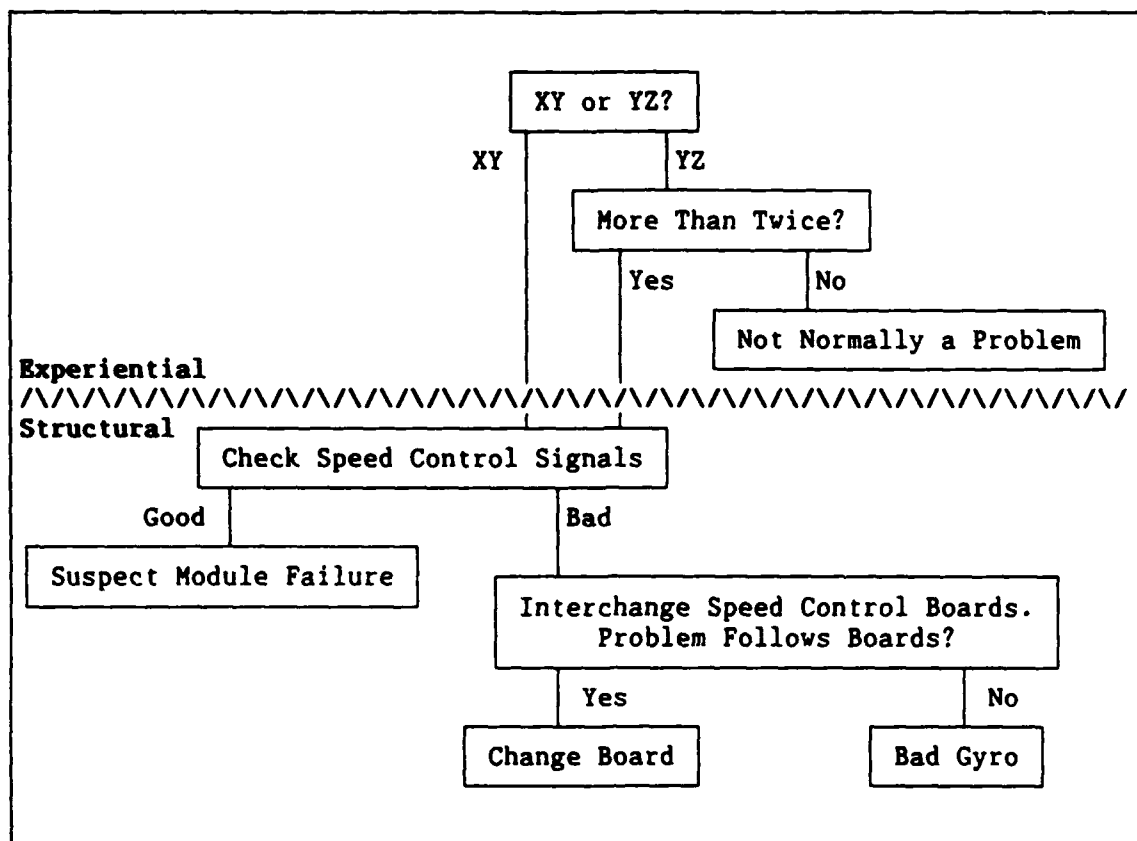


Figure 26. Speed Control Tree Separated into Experiential and Structural Knowledge.

Appendix D is a listing of the code implementing the decision trees in Figures 22 and 24. This code can be divided into three segments; rules that contain experiential knowledge (rules 23, 30-34, 37, 38), rules that contain structural knowledge (rules 26-29, 39, 40, 42-60), and rules that provide a link from the experiential to the structural segment (rules 24, 25, 35, 36, 41).

The blending of the two systems was accomplished by manipulating these rules. The rules containing the experiential knowledge were

preserved; the rules concerning structural knowledge were eliminated (rules 43-60 were preserved only because they are used by other error messages); and the rules providing the link were modified by replacing their conclusion block with:

```
::CONCLUSION action[I] = "Begin Deep Diagnosis."
```

Figure 27 shows the modified version of rule 39. Rule 39 was first seen in Chapter IV, Figure 7.

DEFINE RULE	Rule039
::APPLIED.TO	I:IMU
::PREMISE	check.for.level.platform[I] and azimuth.equal.zero[I]
::CONCLUSION	action[I] = "Begin Deep Diagnosis."

Figure 27. Rule 39 Modified To Initiate Deep Reasoning.

Next, the top level control block of the shallow reasoning system was modified by adding the declaration of a name and subcomponents to the instance of IMU created (both systems share this class instance) and adding the following conditional statement:

```
if action[I] ="Begin Deep Diagnosis  
then invoke deep.diagnosis
```

The resulting control block can be seen in Figure 28. Finally, the top level control block of the deep reasoning system was removed, and the remaining code for the deep reasoning system was appended to the shallow reasoning system.

```

DEFINE CONTROL.BLOCK          ABOUT.IMU
::INVOCATION                  top.level
::TRANSLATION                 "diagnose faults in the IMU "
::BODY
begin
  vars I:imu;
  display spaces(25) ! "THE DMINS FAULT ADVISOR"!
  new.line();
  create.instance imu called I with
  begin
    name.of[I] = "IMU";
    has.subcomponent[I];
    critical.test[I];
    output.test.ok[I]<-1.0>;
  end;
  determine error.message[I];
  determine action[I];
  if action[I] = "Begin deep diagnosis"
  then invoke DIAGNOSE.IMU.FAULT(I)
  else begin
    invoke display.recommendations(I);
    determine fault[I];
  end;
end
END.DEFINE

```

Figure 28. Modified Top Level Control Block.

The resulting system allows both structural and experiential knowledge to be used in diagnosing a problem. It can take advantage of the technician's experience and employ knowledge unrelated to structure. The search generated by the principle of locality, however, does not consider the likelihood of a component's being faulty, nor does it consider the difficulty of testing the component.

Adding Heuristics to the Model

The search generated by the system developed in Phase 3 considers only the principle of locality. This would be acceptable if all components were equally fallible, and equally accessible. However, this is not the case. Based on mean time between failures (MTBF), a gyro is more than seven times as likely to fail as a slip ring. In addition, a slip ring requires 12 times as long to test as a gyro (21). It is desirable to use the principle of locality to identify components that, by their manner of connection, could reasonably be the cause of the fault, but to test the suspected components in an order that will ensure that the most-promising, least-costly tests are performed first and that the least-promising, most-costly tests are performed only as a last resort. This was the goal of the fourth and final phase.

Two attributes were created to be associated with each component in the DMINS IMU model. They are RISK, which is a measure of the likelihood of the component's being at fault based on MTBF, and TEST-COST, which is a measure of the difficulty to test the component based on the time to test (in person-minutes). Engineers at Newark were interviewed to determine RISK and TEST-COST values for each of the components in the model of the DMINS IMU (21). The results are shown in Table 1.

To determine the feasibility of ordering the test, each component in the BDS code was assigned its corresponding value for the attribute RISK. Next, the DEEP.DIAGNOSE control block from Figure 17 was modified to prioritize the search by values of RISK, from highest risk to lowest risk. Code and comments for the control block are in Appendix E. The pseudo code for this algorithm appears in Figure 29.

Table 1. Values for RISK and TEST-COST.

Model	Component	Risk	Test-Cost
Top Level	Velocity Meter Function	Med	Med
	Gyro Speed Control	High	Med
	Gyro Torquing Function	High	Med
	Platform Torquing Function	Low	Med
	Sequence Timing Function	Low	Low
Velocity Meter Function	Test Equipment	Low	Low
	X Velocity Meter	High	Med
	Y Velocity Meter	High	Med
	Frequency Standard Function	Low	Low
	Power Distribution	Low	Low
Gyro Speed Control	DC AMP	Med	Low
	Bandpass Filter	Med	Low
	Gyro Buffer ECA	Med	High
	Gyro	High	Med
	Slip Ring	Low	High
	Power Supply	Low	Low
Gyro Torquing Function	Gyro Torquer	High	Med
	Torque Driver	Med	Low
	Precision Current Network	Med	High

```

Case search level of the current path is:
High:begin
  if Component is not high risk and does not have multiple inputs
  then if Component is an end.point
    then invoke deep.diagnose(Start,Start,Component,Med)
    else designate the input of the Component as Component1 and
        invoke deep.diagnose(Component1,Start,Finish,High)
  else test the output of Component
    if output.test.ok[Component]
    then invoke deep.diagnose(Start,Start,Component,Med)

```

Figure 29. Pseudo Code For the Modified DEEP.DIAGNOSE Control Block.

```

        else begin
            determine if Component has a bad input;
            if Component has a bad input called component1
            then invoke deep.diagnose(component1,component1,finish,high)
            else if Component has subcomponents
            then begin
                build the subcomponents and designate the one
                with the same output as Component as Component2;
                invoke deep.diagnose(Component2,Component2,D,high);
            end
            else the Component is the fault
        end;
    end;
end;

Medium:begin
    if the Risk of Component is low
    then begin
        if Finish = Component
        then invoke deep.diagnose(Start,Start,Finish,Low)
        else invoke deep.diagnose(Component1,Start,Finish,Med)
        end
    else if the output of Component is OK
    then invoke deep.diagnose(Start,Start,Component,Low)
    else if Component has a bad input called Component1
    then invoke deep.diagnose(Component1,Component1,D,Med)
    else if Component has subcomponents
    then begin
        build the subcomponents and designate the one with
        the same output as Component as Component2;
        invoke deep.diagnose(Component2,Component2,D,High);
    end
    else the Component is the fault
    end;

Low:begin
    if Component has a bad input called Component1
    then invoke deep.diagnose(Component1,Component1,Finish,Low)
    else if Component has subcomponents
    then begin
        build the subcomponents and designate the one with the same
        output as Component as Component2;
        invoke deep.diagnose(Component2,Component2,D,High)
    end
    else the Component is the fault
end
end

```

Figure 29. Pseudo Code For the Modified DEEP.DIAGNOSE Control Block (Cont'd).

The algorithm has been enhanced to ensure that the highest risk components (that is, the components most likely to be at fault) are tested first followed by components with medium risk, then those with low risk. Each time the control block is invoked, four arguments are passed; COMPONENT, START, FINISH, and PATH. START is the first component in the current path; the output of START is known to be bad. FINISH is the last component in the current path; the inputs of FINISH are known to be good (initially, FINISH is set to a dummy component since its value has not been determined). PATH designates a Search Level; the Search Level can assume one of three values: high, med, or low.

The program begins with START and COMPONENT set to IMU, FINISH is set to the dummy component, and the Search Level is high. The model for IMU will be constructed, and diagnosis will begin. As long as the Search Level remains high, only components considered critical (those with a high RISK level, or those with multiple inputs) are tested. The output of COMPONENT will be tested first; if it is good, FINISH will be set to this component, and DEEP.DIAGNOSE will be invoked with the same value for START, COMPONENT set to START, and the Search Level set to med. If the output of COMPONENT is bad, the inputs to COMPONENT will be tested. If an input is bad, the component responsible for the bad input will become the new value for START and COMPONENT, FINISH will remain a dummy component and the Search Level will remain high. If, after all critical components are tested, the end of the path is reached without identifying the fault, a medium level search begins with START and COMPONENT set equal to the current value of START, and the endpoint as the value of FINISH. If a component is found with a bad output and all good inputs, it will be tested for subcomponents. If it has subcomponents, they will be built and diagnosis

will resume with a high level search, START and COMPONENT will assume the value of the subcomponent with the same output as the parent component, and FINISH will be set as a dummy component. If no subcomponents exist, COMPONENT is reported as the fault.

During the Medium level search, only medium risk components are tested. If the output of COMPONENT is good, a low level search will begin with the current value of START for START and COMPONENT, and COMPONENT as the new value for FINISH. If the output is bad, the input will be tested. If the input is bad, a medium level search will continue with the component responsible for the bad input as the new value for START and COMPONENT; FINISH will remain set to the same value. If, after all medium RISK components are tested, the end of the path is reached without identifying the fault, a low level search begins with START and COMPONENT set equal to the current value of START, and the endpoint as the value of FINISH. If during the medium level search a component is found to have a bad output and a good input, it will be handled exactly as it is during the high level search.

During the low level search, all components will be tested since only low risk components remain in the path. If during the low level search a component is found to have a bad output and a good input, it will be handled exactly as it is during the high level search.

A component with multiple inputs is considered a critical test point, and for that reason is tested during the high level search regardless of its associated RISK value. This prevents the search from considering branches unnecessarily. If the output of the component is good, the inputs are not considered at all. If the output of the component is bad, the inputs will be tested in order of their respective RISK value, from

highest to lowest. This is accomplished by replacing the single rule shown in Figure 18 of Chapter V with the three ordered rules shown in Figure 30.

```

DEFINE RULE RULE.201
::APPLIED.TO COMPONENT:imu.component
::PREMISE    already.existing(COMPONENT1:imu.component |
                        determined?(connected.to[COMPONENT1,COMPONENT])
                        and connected.to [COMPONENT1,COMPONENT] known
                        and risk[COMPONENT1] is high
                        and output.test.ok[COMPONENT1] thought.not)
::CONCLUSION bad.input[COMPONENT] = connected.to[COMPONENT1,COMPONENT];
END.DEFINE

DEFINE RULE RULE.202
::APPLIED.TO COMPONENT:imu.component
::PREMISE    already.existing(COMPONENT1:imu.component |
                        determined?(connected.to[COMPONENT1,COMPONENT])
                        and connected.to [COMPONENT1,COMPONENT] known
                        and risk[COMPONENT1] is med
                        and output.test.ok[COMPONENT1] thought.not)
::CONCLUSION bad.input[COMPONENT] = connected.to[COMPONENT1,COMPONENT];
END.DEFINE

DEFINE RULE RULE.203
::APPLIED.TO COMPONENT:imu.component
::PREMISE    already.existing(COMPONENT1:imu.component |
                        determined?(connected.to[COMPONENT1,COMPONENT])
                        and connected.to [COMPONENT1,COMPONENT] known
                        and risk[COMPONENT1] is low
                        and output.test.ok[COMPONENT1] thought.not)
::CONCLUSION bad.input[COMPONENT] = connected.to[COMPONENT1,COMPONENT];
END.DEFINE

```

Figure 30. Rules to Order Search of Component's Inputs.

The DEEP.DIAGNOSE control block can be further modified to include consideration of the attribute TEST-COST by increasing the number of legal values for the level of search and adding the necessary code for the case statement in the DEEP.DIAGNOSE control block. Due to the emphasis on

maintainability in this effort, it was determined that the increase in complexity of the program associated with adding TEST-COST outweighed the benefits of this additional prioritizing.

Summary

BDS begins fault diagnosis by applying shallow reasoning in an attempt to isolate the fault. If this attempt fails, BDS constructs a model of the UUT based on structural knowledge. BDS determines which components are possible candidates based on the principle of locality, then orders these components based on the likelihood that a component might fail and the time required to test the component. Finally, BDS uses this information to suggest a sequence of tests to the technician, guiding him adaptively to the location of the fault.

This blend of reasoning-approaches is a better emulation of a technician's behavior than is either of the approaches alone. When a fault occurs a technician tries the repairs he thinks are "obvious." He does not get these from a book or (for the most part) from knowledge of the system, but from his experience that the last several times a situation like this occurred the problem was with component X. Only if the "quick fix" to component X does not solve the problem does the technician resort to the manual to trace the fault back to the component responsible.

VII. Conclusions and Recommendations

Summary

The objective of this thesis was to investigate the use of deep reasoning and shallow reasoning in diagnostic expert systems. This investigation included blending the two type of reasoning in a manner which capitalizes on the strengths of each while diminishing their weaknesses.

The thesis focused on the development of the Blended Diagnostic System (BDS), an expert system which blends the two forms of reasoning by emulating a technician's method of diagnosing faults. BDS begins fault diagnosis by applying shallow reasoning in an attempt to isolate the fault. If this attempt is not successful, BDS constructs a model of the unit under test and diagnoses faults from this model in an order based on the likelihood of the component's being the fault and the difficulty of testing the component.

The system selected to test the performance of BDS is the Dual Miniature Inertial Navigation System (DMINS). Repair of DMINS is the responsibility of the Aerospace Guidance and Metrology Center (AGMC) located at Newark AFB, OH. A prototype of BDS was developed for DMINS in a four phase approach. First, an expert system employing only shallow reasoning was developed using traditional knowledge engineering techniques. Second, an expert system was developed that employs deep reasoning to construct a model of the unit under test (UUT). Third, the shallow and deep reasoning systems were blended. Finally, the deep reasoning portion of the system was enhanced by employing heuristically

guided search-techniques to guide the technician from place to place in the model.

Knowledge acquisition sessions performed for the shallow and deep reasoning portions of BDS differed in that the source of the knowledge for the shallow reasoning system was an expert, whereas the source of the knowledge for the deep reasoning system was a manual, with an expert acting only as an interpreter. Difficulties during the two sessions were similar. In both cases, there was a tendency for the experts to go into more detail than was necessary to build the system. Since time was limited, it was necessary to ensure that discussions remained relevant. The existence of the technical manual in the case of the deep reasoning system was a valuable asset in that it served as a point of focus for the interview. No such focal point existed for the shallow reasoning system, except for the error message written at the top of the page of each decision tree.

Assessment

An assessment of the prototype developed for this thesis reveals several key features as well as a few limitations. Key features of the BDS prototype include:

1. BDS runs on an AT class machine. By developing the system on an AT class personal computer as opposed to a specialized workstation such as a Lisp machine (required by IDM) or a Sun workstation (required by FIS), there was an immediate savings of \$25,000 to AGMC. More importantly, BDS provides a model for similar expert systems to be extended to ATE at Newark AFB and other maintenance organizations,

which, because their primary mission is not research, can not justify the expenditure for specialized work stations.

2. BDS is efficient. The completed code responds to any operator input within ten seconds. The requirement, as stated in Chapter II was ten minutes.
3. BDS is adaptable. As the DMINS system (or any system to which a BDS-like expert system could be applied) ages, different parts become more prone to failure. The attribute Risk allows BDS to take this fact into consideration, and adjust the search accordingly.
4. BDS is equipped with an explanation facility. If the user requests an explanation during the shallow reasoning session, BDS provides an explanation as to why such a test is desirable through the S.1 explanation facility.
5. BDS allows for the addition of a historical database. By building the attribute Fault into the program, a "hook" has been made available that will allow the program to store and retrieve information about previous faults. This will allow the system to detect recurrent problems and act accordingly.
6. BDS allows for integration with the ATE. Using S.1's feature of external functions, attributes that are already in the program can be retrieved directly from the LAN by changing the LEGAL.MEANS slot of the attribute to the name of the external function.
7. BDS has a complete framework. The program already handles all possible error messages through shallow reasoning. The main control block that drives the deep reasoning system is already developed. Three out of ten

functions are complete for the DMINS model. Using these three models as a guide, it will be a relatively simple matter to extend the model for the remaining seven functions.

As mentioned, the BDS prototype has limitations. These limitations include:

1. BDS assumes only a single fault exists. The most restrictive limitation of BDS is that the algorithm employed in the deep reasoning portion of the system assumes that only a single fault exists.
2. Performance of the explanation facility deteriorates during deep reasoning. Due to the recursive nature of the deep reasoning algorithm, requests for explanations during deep reasoning are of limited value.

Conclusions

During research for this thesis, it was noted that while shallow reasoning is a useful method of incorporating an expert's experience with a unit under test (UUT), it is unable to accommodate situations that have not been previously encountered in existing systems. Furthermore, deep reasoning is a useful method for incorporating structural knowledge into an expert system, however, it neglects the technician's experience and does not make use of information that, while not related to structure, is useful for diagnostics (e.g. previous faults, current test).

It was also noted that a model-based search based strictly on the principle of locality does not take into consideration the fact that all components are not equally likely to be the cause of the fault, nor does it consider that component testing has varying levels of difficulty.

It was further observed that technicians use a combination of shallow reasoning, deep reasoning, and heuristically guided search techniques to isolate a fault. First they attempt quick fixes (shallow reasoning), and, if these are not successful they resort to the technical manual (deep reasoning) to identify components that could possibly cause the fault. Finally, they use heuristics based on the likelihood that a component will fail and the difficulty involved in testing the component to determine the order in which the components should be tested.

Based on these observations, and the prototype of the Blended Diagnostic System, the following conclusions can be made:

1. It is possible to blend deep and shallow reasoning by initially applying the technician's experience, and if not successful, resorting to structural diagnosis to isolate a fault.
2. The performance of the deep reasoning portion of such a blended system can improved by employing heuristically-guided search techniques rather than relying solely on the principle of locality.
3. The resulting blend of reasoning techniques is a better emulation of a technician's method of diagnosing a fault than is either shallow or deep reasoning applied alone.
4. This method of blending shallow and heuristically-guided deep reasoning extends the class of diagnosable faults beyond the class for either form of reasoning alone, and reduces the number and duration of tests to be performed by the technician.
5. Use of a document as a knowledge source does not necessarily eliminate the need for an expert during knowledge acquisition. Consequently, many of the problems associated with interviewing remain. The document does, however, provide a point of focus for discussion, thereby providing a

convenient point to which the knowledge engineer can return when discussions degrade to an unproductive state.

Recommendations

Specific areas for further research into the area of blending deep and shallow reasoning include generic improvements to BDS (recommendations 1-3) as well as improvements to BDS as it applies to DMINS (recommendations 4-8).

1. Develop a simple graphic display to interface with BDS. This display, similar to that of FIS, would be a block diagram of the components of the system. Each block would be color-coded to indicate its current status (e.g., red - faulty, yellow - suspect, green - good, orange - not as yet tested).
2. Separate the knowledge bases from the control mechanisms to develop a shell that will allow the knowledge engineer to use BDS as an aid to developing an expert system to work with any ATE. This shell could then be tested on an existing Air Force System.
3. Extend the reasoning algorithm in the deep reasoning portion to allow for the possibility of multiple faults.
4. Complete the model for the deep reasoning portion of the code to include all ten of the functions of the IMU.
5. Interface the program to directly receive the parameters available on the local area network at AGMC. This should start with an attempt to retrieve the error message directly, and then be expanded to include all possible parameters.

6. Extend the rules and models in BDS so that it is capable of diagnosing faults during Mode B tests.
7. After the system grows to a size such that it has twice the number of its current rules, implement the rule-categories discussed in Chapter IV and determine if efficiency is increased.
8. Interface the system to the DMINS historical database, when such a database is developed by the engineers at Newark AFB.

Appendix A: Mode A Testing

The expert system operation is limited to Mode A tests. This appendix describes the sequence of testing during a typical Mode A test.

1. Shim Cal - Tests velocity meters. First pass compares current numbers of bias and scale factor with numbers obtained during last test. If numbers are within a predefined error margin the next test begins. Otherwise this test is repeated. If it passes, continue next test. Otherwise halt and report failure. Up to four hours required.
2. Gyro Cal - Tests Gyros. Up to three passes can be run. If any are passed, test continues. Otherwise failure is reported. Requires up to eighteen hours.
3. Self Align - Tests ability of IMU to perform a self alignment. Requires up to four hours.
4. Nav Align - Tests the ability of the IMU to navigate. Requires up to sixteen hours.
5. Navigation - Further test of the IMU's ability to navigate correctly. Requires thirty hours for complete navigation test.

Appendix B: The 62 Error Messages

The DMINS ATE is capable of generating 62 possible error messages.

BDS uses these error messages to initiate diagnosis. The error messages are listed below.

- | | |
|------------------------------|---------------------------------------|
| 1. velocity.unreasonable | 32. minor.reset.fault |
| 2. vt.greater.than.2.knots | 33. minor.fault.cont |
| 3. xy.speed.control | 34. imu.minor |
| 4. yz.speed.control | 35. plat.stab.abort |
| 5. imu.major | 36. xvm.precounter.fault |
| 6. no.input.3.axes | 37. yvm.precounter.fault |
| 7. no.input.az | 38. both.vm.precounter.failure |
| 8. no.input.pitch | 39. vm.bite.failure |
| 9. no.input.roll | 40. x.gyro.torque.fault |
| 10. automatic.shutdown | 41. y.gyro.torque.fault |
| 11. imu.o.load | 42. z.gyro.torque.fault |
| 12. imu.o.temp | 43. system.not.properly.caged |
| 13. pwr.interrupt | 44. gyro.hot |
| 14. dcc.o.load.o.temp | 45. gyro.cold |
| 15. dcc.o.temp | 46. gyro.temp.normal |
| 16. i.c.fault | 47. mux.decoder.dl.fault |
| 17. comp.tie.in.sw.on | 48. cage.xy.dl.fault |
| 18. i.c.fault.inhb.enab | 49. cage.yz.dl.fault |
| 19. seq.cnt.no.compare | 50. gyro.start.dl.fault |
| 20. i.c.data.loop.fault | 51. gyro.run.dl.fault |
| 21. i.c.fault.cont | 52. uyk.good.dl.fault |
| 22. in.parity.test.inhb.enab | 53. input.parity.dl.fault.mux04 |
| 23. out.word.par.inhb.enab | 54. input.parity.no.2.dl.fault |
| 24. input.parity.fault | 55. output.word.parity.dl.fault.mux09 |
| 25. output.word.parity.fault | 56. vt.vr.greater.than.3.knots |
| 26. output.word.parity.cont | 57. minisins.vel.dif.exceeds.limit |
| 27. excess.angle | 58. minisins.pos.dif.exceeds.limit |
| 28. servo.disable | 59. parity.test.1.no.go |
| 29. major.reset.fault | 60. parity.test.2.no.go |
| 30. z.stab | 61. parity.test.3.no.go |
| 31. system.in.free.run | 62. put.intercom.test.no.go |

Appendix C: The 38 Shop Repairable Units (SRU)

This appendix lists the 38 individual shop repairable units (SRU) comprised by the Inertial Measuring Unit. The goal of the expert system is to isolate an IMU fault to one or more of the SRUs listed.

<u>ID#</u>	<u>Name</u>
3A1	Bandpass Filter and Shift Register
3A2	Bandpass Filter and Shift Register
3A3	Precision Torquing Driver (X)
3A4	Precision Torquing Driver (Y)
3A5	Precision Torquing Driver (Z)
3A7	Platform Electronic Switch
3A8	Shorting Plug
3A9	Precision Current Network
3A10	Stable Platform
3A10A3	Displacement Gyroscope (X-Y)
3A10A4	Displacement Gyroscope (Y-Z)
3A10A7	Velocity Meter (X)
3A10A8	Velocity Meter (Y)
3A10AR1	Resolver Buffer Amp
3A10AR5	Gyro Buffer Amp (X-Y)
3A10AR6	Gyro Buffer Amp (Y-Z)
3PS1	640 Hz Power Supply (X-Y)
3PS2	640 Hz Power Supply (Y-Z)
3PS3	Power Cube
3PS7	400 Hz Power Supply No.2
3PS8	400 Hz Power Supply No.1
3PS9	Triangle Generator and Case Rotation Power Supply
3PS10	4.8 KHz Power Supply
3PS11	Frequency Standard
3AR1	D.C. Amplifier (X-Y)
3AR2	D.C. Amplifier (Y-Z)
3AR3	Synchro Signal Buffer Amp
3AR4	Gyro Cage Amp
3AR5	Thermoelectric Signal Amp.
3AR6	Gyro Temperature Controller
3AR7	Gimbal Cage Amp.
3AR8	Platform Signal Amp
3AR9	Platform Electronic Control Amp (Roll)
3AR10	Platform Electronic Control Amp (Pitch)
3AR11	Platform Electronic Control Amp (Azimuth)
3AR12	Gimbal Rate Electronic Control Amp (Roll)
3AR13	Gimbal Rate Electronic Control Amp (Pitch)
3AR14	Gimbal Rate Electronic Control Amp (Azimuth)

**Appendix D: Rules for XY Speed Control, YZ Speed Control,
Velocity Unreasonable, and Vt Greater Than 2 Knots**

```

/*****
/*
/* Rules 023 - 029 determines the cause of the following messages:
/*
/*          xy.speed.control,
/*          yz.speed.control.
/*
DEFINE RULE rule023
  ::APPLIED.TO I:IMU
  ::PREMISE    error.message[I] is yz.speed.control and
               not yz.fault.occurred.more.than.twice[I]
  ::CONCLUSION begin
               action[I] = "No action is required. If the YZ speed
               control error has not occurred more than twice, it is
               normally not a problem";
               fault[I] = none;
  end
END.DEFINE

DEFINE RULE rule024
  ::APPLIED.TO I:IMU
  ::PREMISE    error.message[I] is xy.speed.control
  ::CONCLUSION check.speed.control.signal[I]
END.DEFINE

DEFINE RULE rule025
  ::APPLIED.TO I:IMU
  ::PREMISE    error.message[I] is yz.speed.control and
               yz.fault.occurred.more.than.twice[I]
  ::CONCLUSION check.speed.control.signal[I]
END.DEFINE

DEFINE RULE rule026
  ::APPLIED.TO I:IMU
  ::PREMISE    check.speed.control.signal[I] and
               speed.control.signal.good[I]
  ::CONCLUSION action[I] = "Check the discretes on the modules. A
               module problem is suspected"
END.DEFINE

DEFINE RULE rule027
  ::APPLIED.TO I:IMU
  ::PREMISE    check.speed.control.signal[I] and
               not speed.control.signal.good[I]
  ::CONCLUSION exchange.speed.control.cards[I]
END.DEFINE

```

```

DEFINE RULE rule028
::APPLIED.TO I:IMU
::PREMISE      exchange.speed.control.cards[I] and
                not problem.follows.speed.control.card[I]
::CONCLUSION   begin
                replace.faulty.component[I];
                fault[I] = corresponding.gyro;
                end
END.DEFINE

```

```

DEFINE RULE rule029
::APPLIED.TO I:IMU
::PREMISE      exchange.speed.control.cards[I] and
                problem.follows.speed.control.card[I]
::CONCLUSION   begin
                replace.faulty.component[I];
                fault[I] = corresponding.speed.card;
                end
END.DEFINE

```

```

/*****
/*
/* This group of rules includes rules 030 - 042 and troubleshoots the */
/* following two error messages:                                     */
/*          velocity.unreasonable,                                  */
/*          vt.greater.than.2.knots.                               */
/* In addition, this section uses the routine Check Gyro Circuit, which */
/* starts with rule 043.                                           */
*/

```

```

DEFINE RULE rule030
::APPLIED.TO I:IMU
::PREMISE      error.message[I] is in {velocity.unreasonable,
                vt.greater.than.2.knots) and
                test[I] is gyro.cal
::CONCLUSION   check.position[I]
END.DEFINE

```

```

DEFINE RULE rule031
::APPLIED.TO I:IMU
::PREMISE      error.message[I] is in {velocity.unreasonable,
                vt.greater.than.2.knots) and
                test[I] is navigate
::CONCLUSION   check.velocity.direction[I]
END.DEFINE

```

```

DEFINE RULE rule032
::APPLIED.TO I:IMU
::PREMISE      error.message[I] is in {velocity.unreasonable,
                                         vt.greater.than.2.knots} and
                                         not test[I] is in {gyro.cal, navigate}
::CONCLUSION  action[I]= "See the shop supervisor. Normally this message
                           occurs only when the IMU is in gyro.cal or
                           navigate"

```

END.DEFINE

```

DEFINE RULE rule033
::APPLIED.TO I:IMU
::PREMISE      check.position[I] and
north.south.or.east.west[I]
::CONCLUSION  wait.for.platform.to.slew[I]
END.DEFINE

```

```

DEFINE RULE rule034
::APPLIED.TO I:IMU
::PREMISE      wait.for.platform.to.slew[I] and
velocities.changed.directions[I]
::CONCLUSION  begin
               replace.faulty.component[I];
               fault[I] = corresponding.velocity.meter;
               end

```

END.DEFINE

```

DEFINE RULE rule035
::APPLIED.TO I:IMU
::PREMISE      wait.for.platform.to.slew[I] and
not velocities.changed.directions[I]
::CONCLUSION  check.gyro.circuit[I]
END.DEFINE

```

```

DEFINE RULE rule036
::APPLIED.TO I:IMU
::PREMISE      check.position[I] and
not north.south.or.east.west[I]
::CONCLUSION  check.gyro.circuit[I]
END.DEFINE

```

```

DEFINE RULE rule037
::APPLIED.TO I:IMU
::PREMISE      check.velocity.direction[I] and
north.south.or.east.west[I]
::CONCLUSION  begin
               replace.faulty.component[I];
               fault[I] = corresponding.velocity.meter;
               end
END.DEFINE

```

DEFINE RULE rule038

```
::APPLIED.TO I:IMU
::PREMISE    check.velocity.direction[I] and
              not north.south.or.east.west[I]
::CONCLUSION check.for.level.platform[I]
END.DEFINE
```

DEFINE RULE rule039

```
::APPLIED.TO I:IMU
::PREMISE    check.for.level.platform[I] and
              not azimuth.equal.zero[I]
::CONCLUSION check.velocity.meter[I]
END.DEFINE
```

DEFINE RULE rule040

```
::APPLIED.TO I:IMU
::PREMISE    check.velocity.meter[I] and
              faulty.velocity.meter[I]
::CONCLUSION begin
              replace.faulty.component[I];
              fault[I] = corresponding.velocity.meter;
              end
END.DEFINE
```

DEFINE RULE rule041

```
::APPLIED.TO I:IMU
::PREMISE    check.for.level.platform[I] and
              azimuth.equal.zero[I]
::CONCLUSION check.gyro.circuit[I]
END.DEFINE
```

DEFINE RULE rule042

```
::APPLIED.TO I:IMU
::PREMISE    check.velocity.meter[I] and
              not faulty.velocity.meter[I]
::CONCLUSION check.gyro.circuit[I]
END.DEFINE
```

```
/*****
/*
/*          CHECK GYRO CIRCUIT ROUTINE          */
/* This routine is used for several different error messages including */
/* velocity unreasonable, vt. greater than 2 knots, imu major, and    */
/* plat stab abort.                                                    */
```

DEFINE RULE rule043

```
::APPLIED.TO I:IMU
::PREMISE    check.gyro.circuit[I] and
              speed.control.signal.good[I]
::CONCLUSION check.gyro.run.voltage[I]
END.DEFINE
```

```

DEFINE RULE rule044
  ::APPLIED.TO I:IMU
  ::PREMISE    check.gyro.run.voltage[I] and
                gyro.run.voltage.good[I]
  ::CONCLUSION check.pick.off.signals[I]
END.DEFINE

DEFINE RULE rule045
  ::APPLIED.TO I:IMU
  ::PREMISE    check.pick.off.signals[I] and
                pick.off.signals.good[I]
  ::CONCLUSION check.pick.off.signals.while.gyros.not.running[I]
END.DEFINE

DEFINE RULE rule046
  ::APPLIED.TO I:IMU
  ::PREMISE    check.pick.off.signals[I] and
                not pick.off.signals.good[I]
  ::CONCLUSION begin
                  replace.faulty.component[I];
                  fault[I] = corresponding.gyro;
                end
END.DEFINE

DEFINE RULE rule047
  ::APPLIED.TO I:IMU
  ::PREMISE    check.pick.off.signals.while.gyros.not.running[I] and
                not pick.off.signals.while.gyros.not.running.good[I]
  ::CONCLUSION begin
                  replace.faulty.component[I];
                  fault[I] = corresponding.gyro;
                end
END.DEFINE

DEFINE RULE rule048
  ::APPLIED.TO I:IMU
  ::PREMISE    check.pick.off.signals.while.gyros.not.running[I] and
                pick.off.signals.while.gyros.not.running.good[I]
  ::CONCLUSION monitor.speed.control.for.b14.and.up[I]
END.DEFINE

DEFINE RULE rule049
  ::APPLIED.TO I:IMU
  ::PREMISE    monitor.speed.control.for.b14.and.up[I] and
                speed.control.for.b14.and.up.good[I]
  ::CONCLUSION run.next.test[I]
END.DEFINE

```

```

DEFINE RULE rule050
  ::APPLIED.TO I:IMU
  ::PREMISE    monitor.speed.control.for.b14.and.up[I] and
               not speed.control.for.b14.and.up.good[I]
  ::CONCLUSION begin
               replace.faulty.component[I];
               fault[I] = corresponding.gyro;
               end
END.DEFINE

DEFINE RULE rule051
  ::APPLIED.TO I:IMU
  ::PREMISE    check.gyro.run.voltage[I] and
               not gyro.run.voltage.good[I]
  ::CONCLUSION interchange.ps1.and.ps2[I]
END.DEFINE

DEFINE RULE rule052
  ::APPLIED.TO I:IMU
  ::PREMISE    interchange.ps1.and.ps2[I] and
               problem.follows.power.supply.card[I]
  ::CONCLUSION begin
               replace.faulty.component[I];
               fault[I] = corresponding.power.supply.card;
               end
END.DEFINE

DEFINE RULE rule053
  ::APPLIED.TO I:IMU
  ::PREMISE    interchange.ps1.and.ps2[I] and
               not problem.follows.power.supply.card[I]
  ::CONCLUSION begin
               replace.faulty.component[I];
               fault[I] = corresponding.gyro;
               end
END.DEFINE

DEFINE RULE rule054
  ::APPLIED.TO I:IMU
  ::PREMISE    check.gyro.circuit[I] and
               not speed.control.signal.good[I]
  ::CONCLUSION interchange.speed.control.cards[I]
END.DEFINE

```

DEFINE RULE rule055

```
::APPLIED.TO I:IMU
::PREMISE interchange.speed.control.cards[I] and
not problem.follows.speed.control.card[I]
::CONCLUSION begin
replace.faulty.component[I];
fault[I] = corresponding.gyro;
end
```

END.DEFINE

DEFINE RULE rule056

```
::APPLIED.TO I:IMU
::PREMISE interchange.speed.control.cards[I] and
problem.follows.speed.control.card[I]

::CONCLUSION begin
replace.faulty.component[I];
fault[I] = corresponding.speed.card;
end
```

END.DEFINE

DEFINE RULE rule057

```
::APPLIED.TO I:IMU
::PREMISE run.next.test[I] and
not next.test.good[I]
::CONCLUSION check.speed.stability[I]
```

END.DEFINE

DEFINE RULE rule058

```
::APPLIED.TO I:IMU
::PREMISE run.next.test[I] and
next.test.good[I]
::CONCLUSION run.scorsby.test[I]
```

END.DEFINE

DEFINE RULE rule059

```
::APPLIED.TO I:IMU
::PREMISE run.scorsby.test[I] and
not scorsby.good[I]
::CONCLUSION check.speed.stability[I]
```

END.DEFINE

DEFINE RULE rule060

```
::APPLIED.TO I:IMU
::PREMISE run.scorsby.test[I] and
scorsby.good[I]
::CONCLUSION begin
action[I] = "Testing is complete. All inputs indicate
this testing is good";
fault[I] = none;
end
```

END.DEFINE

Appendix E: The Deep Diagnose Control Block

```

/*****
/*
/*          DEEP DIAGNOSE
/*
/* This Control Block is the Inference Engine of this portion of the
/* program. Although it is based on a program from the S.1 manual, it
/* has been greatly enhanced. The control block determines the fault
/* of the system by searching for a bad input to the current system.
/* If one is found, the control block is called with the component
/* responsible for the bad input as its argument. This continues until
/* a component is found with a bad output, but no bad inputs. It is
/* then determined if this component has subcomponents. If so, the
/* subcomponents are built and the subcomponents are diagnosed. When
/* a component with a bad output, good inputs, and no subcomponents,
/* the component is identified as the faulty component and the search
/* is terminated. This algorithm has been enhanced to ensure that the
/* highest risk components (that is, the components most likely to be
/* at fault) are tested first followed by components with medium risk,
/* then those with low risk. In the case of components with multiple
/* inputs, the high risk inputs will be tested first. This is due to
/* the ordering of the Rules that cause the components to be tested.
/* The Arguments for this control block are:
/*
/*   Component - The component currently under consideration.
/*   Start - The first component in the current path being tested.
/*   Finish - The last component in the current path being tested.
/*             (Initially, a dummy variable since the end is not known)
/*   P:Path - The level of the current search. This dictates whether a
/*             component will be tested immediately. Initially, the
/*             search level of the path will be set to high level and
/*             only components with high risk or multiple inputs will
/*             be tested. If the fault is not found, the search level
/*             will be lowered to medium level, then finally low level.
*/

```

DEFINE CONTROL.BLOCK

::INVOCATION

::ARGUMENTS

DEEP.DIAGNOSE

internal

COMPONENT:imu.component,

START:imu.component,

FINISH:imu.component,

p:path

::BODY

begin

display new.line()!

new.line()! "Deep Diagnose was invoked with"!

new.line()! "component = " ! instance.trans(component)!

new.line()! "start = " ! instance.trans(start) !

new.line()! "finish = " ! instance.trans(finish)!

new.line()! "level = " ! search.level[p];

Case search.level[p] of
high.level:

/* In a high level search end will always equal 0 because as soon as an
end

point is found, level of search will be changed to medium. Also, the output of start is always bad. A critical test component is defined as one that is high risk or has multiple inputs. The pseudo code for high level is

```
if not critical.test(component) and not end.point(component)
then call deep.diagnose(component1,start,0,high)
```

```
if not critical.test(component) and end.point(component)
then call deep.diagnose(start,start,component,med)
```

```
if critical.test(component) and output.test.ok(component)
then call deep.diagnose(start,start,component,med)
```

```
if critical.test(component) and not output.test.ok(component)
then determine bad.input[component]
```

```
    if bad.input exists call
    deep.diagnose(component1,component1,0,high)
    if bad.input does not exist determine if component
    has subcomponents. If so, call
    deep.diagnose(component2,component2,0,high)
    if component does not have subcomponents create
    instance fault with name.of[component].      */
```

```
begin
```

```
    determine critical.test[COMPONENT];
```

```
    if not critical.test[COMPONENT]
```

```
    then begin
```

```
        if end.point[COMPONENT]
```

```
        then begin
```

```
            if exists(med:path|search.level[med] is
            med.level)
```

```
            then invoke deep.diagnose(start,start,component,med)
```

```
            end
```

```
        else begin
```

```
            if exists(COMPONENT1:imu.component,
```

```
                high:path|connected.to[COMPONENT1,COMPONENT] known
                and search.level[high] is high.level)
```

```
            then invoke deep.diagnose(component1,start,finish,high)
```

```
            end
```

```
        end
```

```
    else begin
```

```
        determine output.test.ok[component];
```

```
        if output.test.ok[component]
```

```
        then begin
```

```
            if exists(med:path|search.level[med] is med.level)
```

```
            then invoke deep.diagnose(start,start,component,med)
```

```
            end
```

```
        else begin
```

```
            determine bad.input[COMPONENT];
```

```
            if bad.input[COMPONENT] definite
```

```

        then begin
            if exists(COMPONENT1:imu.component,high:path|
                connected.to[COMPONENT1,COMPONENT] =
                bad.input[COMPONENT] and
                search.level[high] is high.level)
            then invoke
                deep.diagnose(component1,component1,finish,high)
            end
        else begin
            if has.subcomponent[COMPONENT]
            then begin
                invoke build.subcomponent(COMPONENT);
                if exists (COMPONENT2:imu.component,
                    high:path,d:dummy|
                    same.output.as[COMPONENT,COMPONENT2]
                    and search.level[high] is high.level)
                then begin
                    invoke
                        deep.diagnose(COMPONENT2,COMPONENT2,d,high);
                end
            end
            else display new.line()! "The fault was found"!
                " to be the " ! instance.name(COMPONENT);
            end;
        end;
    end;
    end;

```

med.level:

```

/* There are two ways med is called from the high level.
First, if the end.point is found so that the fault has been
isolated between either start and end.point or a high and an
endpoint. Second if it has been isolated between two
endpoints. Either way, a start and an end are known. Also,
the output of start is known to be bad. End is never 0. There
will be no high risk or multiple input components in the
path. */

```

```

/* if risk(component) is low and not component = end then
call deep diagnose(component1,start,end,med) */

```

```

/* if risk(component) is low and component = end then
call deep diagnose(start,start,end,low) */

```

```

/* if risk(component) is med and output.test.ok(component)
then call deep.diagnose(start,start,component,low) */

```

```

/* if risk(component) is med and not output.test.ok(component)
then determine bad.input[component] */

```

```

/* if bad.input exists call
deep.diagnose(component1,component1,0,med)*/

```

```

/* if bad.input does not exist determine if component
has subcomponents. If so, call
deep.diagnose(component2,component2,0,high) */

/* if component does not have subcomponents create
instance fault with name.of[component]. */

begin
  if risk[component] is low
  then begin
    if finish = component
    then begin
      if exists(low:path|search.level[low] is low.level)
      then invoke deep.diagnose(start,start,finish,low)
      end
    else begin
      if exists(COMPONENT1:imu.component,med:path|
        connected.to[COMPONENT1,COMPONENT] known and
        search.level[med] is med.level)
      then invoke deep.diagnose(component1,start,finish,med)
      end
    end
  else if output.test.ok[COMPONENT]
  then begin
    if exists(low:path|search.level[low] is low.level)
    then invoke deep.diagnose(start,start,component,low)
    end
  else begin
    determine bad.input[COMPONENT];
    if bad.input[COMPONENT] definite
    then begin
      if exists(COMPONENT1:imu.component,d:dummy,
        med:path|connected.to[COMPONENT1,COMPONENT] =
        bad.input[COMPONENT] and search.level[med]
        is med.level)
      then invoke deep.diagnose(component1,component1,d,med)
      end
    else begin
      if has.subcomponent[COMPONENT]
      then begin
        invoke build.subcomponent(COMPONENT);
        if exists (COMPONENT2:imu.component,
          d:dummy,high:path|
          same.output.as[COMPONENT,COMPONENT2]
          and search.level[high] is high.level)
        then invoke
          deep.diagnose(COMPONENT2,COMPONENT2,d,high)
        end
      else display new.line()! "The fault was found"!
        " to be the " ! instance.name(COMPONENT);
      end
    end
  end
end

```

```

end;

low.level:
/* risk is known to be low.*/
/* input known to be single.*/
/* the output is bad */

/* if bad.input exists call
   deep.diagnose(component1,component1,0,low) */

/* if bad.input does not exist determine if component
   has subcomponents. If so, call
   deep.diagnose(component2,component2,0,high) */

/* if component does not have subcomponents create
   instance fault with name.of[component]. */

begin
  determine bad.input[COMPONENT];
  if bad.input[COMPONENT] definite
  then begin
    if exists(COMPONENT1:imu.component,low:path|
      connected.to[COMPONENT1,COMPONENT] =
      bad.input[COMPONENT] and search.level[low] is
      low.level)
    then invoke deep.diagnose(component1,component1,finish,low)
    end
  else begin
    if has.subcomponent[COMPONENT]
    then begin
      invoke build.subcomponent(COMPONENT);
      if exists (COMPONENT2:imu.component,d:dummy,
        high:path|same.output.as[COMPONENT,COMPONENT2]
        and search.level[high] is high.level)
      then invoke deep.diagnose(COMPONENT2,COMPONENT2,d,high)
      end
    else display new.line()! "The fault was found"!
      " to be the " ! instance.name(COMPONENT);
    end
  end
end
end;
end;
END.DEFINE

```

Appendix F: Code For The Blended Diagnostic System

```
/*                      The Blended Diagnostic System (BDS)                      */
/*                                                                */
/* Coder: 1Lt Jim Skinner                                         */
/* Last Update: 23 Aug 88                                         */
/*                                                                */
/* Description: The Blended Diagnostic System BDS has been designed to */
/* isolate faults in the Dual Miniature Inertial Navigation System */
/* (DMINS) Inertial Measurement Unit (IMU). The expert system begins by */
/* querying the user for the error message reported by the automatic */
/* test equipment (ATE), and then guides the technician through the */
/* tests necessary to determine the fault in the IMU. There are a total */
/* of 62 possible error messages that can be reported by the ATE. The */
/* possible faults are the 38 shop replaceable units contained in the */
/* IMU.                                                            */
/* The knowledge that BDS uses to determine the fault is from two */
/* sources; the first is the shallow knowledge collected from the */
/* expert technician during knowledge engineering sessions. The second */
/* source of knowledge is the deep knowledge obtained by constructing a */
/* model of the system based on structural knowledge contained in the */
/* TOS.                                                            */
/* Status: The code for BDS was written with an emphasis on */
/* maintainability so that the system could continue to grow after */
/* final delivery. The shallow portion of the code covers all of the */
/* 62 error messages. The Deep portion of the code is complete for four */
/* of these messages. They are: xy speed control, yz speed control, */
/* velocity unreasonable, and vt greater than 2 knots. The intent of */
/* the coder is that this system will grow to be complete in both */
/* shallow and deep knowledge. */
/*                                                                */
/* This system was developed in S.1. The structure of the code is as */
/* follows:                                                         */
/*                                                                */
/* I. Shallow Reasoning                                           */
/*   a. Control Blocks                                           */
/*   b. Classes                                                  */
/*   c. Attributes *                                             */
/*   d. Value Hierarchies                                         */
/*   e. Rules **                                                  */
/*                                                                */
/* II. Deep Reasoning                                             */
/*   a. Control Blocks                                           */
/*   b. Classes *                                                */
/*   c. Attributes *                                             */
/*   d. Value Hierarchies                                         */
/*   e. Rules                                                     */
/*                                                                */
/*   * In alphabetical order.                                     */
/*                                                                */
/* ** Rules 25, 35, 36, 38, and 41 provide the transition from */
/* shallow to deep reasoning.                                     */
```

```

/*****
/*
/*          Part I. Shallow Reasoning          */
/*
/* The main attributes determined by this program are:
/*
/*
/* 1. Fault: Fault is the component that the expert system determined
/*    as responsible for the error message received. Fault is
/*    determined (in some cases) after action is recommended. This
/*    forces Fault to default to not.determined if no rule concludes
/*    its value. Fault allows an operator to query S.1 about previous
/*    faults in an IMU and provides a "hook" for future programs to
/*    store previous faults in a historical database through the use
/*    of S.1 external functions.
/*
/*
/* 2. Action: Action is the recommended course of action for the
/*    operator to take in order to correct the fault.
/*
/*****
/*
/*          CONTROL BLOCK DEFINITIONS          */
/*
/*
/* The first Control Block is the top.level control block invoked first*/
/* when a consultation is started
*/

```

```

DEFINE CONTROL.BLOCK          ABOUT.IMU
::INVOCATION                  top.level
::TRANSLATION                  "diagnose faults in the IMU "
::BODY
  begin
    vars I:imu;
    display spaces(25) ! "THE DMINS FAULT ADVISOR"!
      new.line();
    create.instance imu called I with
      begin
        name.of[I] = "IMU";
        has.subcomponent[I];
        critical.test[I];
        output.test.ok[I]<-1.0>;
      end;
    determine error.message[I];
    determine action[I];
    if action[I] = "Begin deep diagnosis"
    then invoke DIAGNOSE.IMU.FAULT(I)
    else
      begin
        invoke display.recommendations(I);
        determine fault[I];
      end;
    end
  end
END.DEFINE

```

```

/*****/

/* An internal control block used to display the values concluded for */
/* the action attribute */

DEFINE CONTROL.BLOCK      display.recommendations
  ::INVOCATION            internal
  ::ARGUMENTS             I:IMU
  ::TRANSLATION            "Display recommended action "
  ::BODY
    begin
      display new.line() ! capitalize(Action[I]) ! ".";
    end
END.DEFINE

/*****/

/*                                CLASSES                                */
/* "IMU" is the only class in the shallow reasoning system.           */
/* */

DEFINE CLASS              IMU
  ::NUMBER.INSTANCES      1
  ::PRINT.ID              "IMU #"
  ::CLASS.TRANSLATION      "an IMU "
  ::PLURAL.CLASS.TRANSLATION "the IMUs "
  ::BLAND.INSTANCE.TRANSLATION "the IMU "
  ::FULL.INSTANCE.TRANSLATION "this IMU"
  ::ANNOUNCEMENT          new.line() ! indent() ! "This IMU will be
                           referred to as: " !
                           instance.name(IMU) ! new.line() ! outdent()
END.DEFINE

/*****/

/*                                MAIN ATTRIBUTES                        */
/* The following attributes are the main attributes in the program.    */

DEFINE ATTRIBUTE          error.message
  ::DEFINED.ON            IMU
  ::TYPE                  text
  ::MULTIVALUED           false
  ::LEGAL.VALUES          error.messages
  /* Legal values use error.messages value hierarchy */
  ::LEGAL.MEANS            {query.user}
  ::DETERMINATION.MEANS    {query.user}
  ::PROMPT                 "What error message is displayed ?"
  ::TRANSLATION            "the displayed IMU Error Message"
END.DEFINE

```

```

DEFINE ATTRIBUTE          fault
::DEFINED.ON             IMU
::TYPE                   text
::MULTIVALUED            false
::LEGAL.VALUES           possible.faults
/* Legal values use faults value hierarchy */
::LEGAL.MEANS             {try.rules}
::DEFAULT.VALUES         {not.determined}
::DETERMINATION.MEANS    {try.rules}
::TRANSLATION            "the fault of the problem"
END.DEFINE

```

```

DEFINE ATTRIBUTE          action
::DEFINED.ON             imu
::TYPE                   text
::MULTIVALUED            false
::LEGAL.MEANS            {try.rules}
::DETERMINATION.MEANS    {try.rules}
::TRANSLATION            "the recommendations for fixing the problem"
END.DEFINE

```

```

/*****
/*
/*                      REMAINING ATTRIBUTES                      */
/*
/*  The remaining attributes are listed in alphabetical order.    */
/*
*/

```

```

DEFINE ATTRIBUTE          able.to.restart
::DEFINED.ON             imu
::TYPE                   boolean
::LEGAL.MEANS            {query.user}
::DETERMINATION.MEANS    {query.user}
::PROMPT                 "Install the diagnostic cap and attempt to
                          restart the test. Is a restart possible?"
::TRANSLATION            "all three axes " ! verb("are", "are not") !
                          " near zero"

```

END.DEFINE

```

DEFINE ATTRIBUTE          all.three.axes.near.zero
::DEFINED.ON             imu
::TYPE                   boolean
::LEGAL.MEANS            {query.user}
::DETERMINATION.MEANS    {query.user}
::PROMPT                 "Check the A to D converter. Are all three
                          axes near zero?"
::TRANSLATION            "all three axes " ! verb("are", "are not") !
                          " near zero"

```

END.DEFINE

```

DEFINE ATTRIBUTE
::DEFINED.ON
::TYPE
::LEGAL.VALUES
::LEGAL.MEANS
::DETERMINATION.MEANS
::PROMPT

::TRANSLATION
END.DEFINE

DEFINE ATTRIBUTE
::DEFINED.ON
::TYPE
::LEGAL.MEANS
::DETERMINATION.MEANS
::TRANSLATION
END.DEFINE

DEFINE ATTRIBUTE
::DEFINED.ON
::TYPE
::LEGAL.MEANS
::DETERMINATION.MEANS
::PROMPT

::TRANSLATION
END.DEFINE

DEFINE ATTRIBUTE
::DEFINED.ON
::TYPE
::LEGAL.MEANS
::DETERMINATION.MEANS
::PROMPT

::TRANSLATION
END.DEFINE

angle.occurs.first.on
imu
text
{xy.pickoff, yz.pickoff}
{query.user}
{query.user}
"Monitor test points 77, 79, and 81 (gyro
pickoffs). Does the excess angle occur first
on the XY gyro pickoff or the YZ gyro
pickoff?"
"the excess angle "

attempt.restart
imu
boolean
{try.rules}
{try.rules}
"attempt a restart"

azimuth.equal.zero
imu
boolean
{query.user}
{query.user}
"Check the A-D converter on the test station.
Is the platform level (azimuth = 0)?"
"the platform " ! verb(" is", "is not")
! " level"

cables.good
imu
boolean
{query.user}
{query.user}
"Check cables J1 and J2. Are the cables
good?"
"cables J1 and J2 " ! verb("are", "are
not") ! "good"

```

DEFINE ATTRIBUTE	case.rotation.normal
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{query.user}
::DETERMINATION.MEANS	{query.user}
::PROMPT	"Pull the gyro power supply cards. DO NOT TURN THE IMU OFF! Next, put the diagnostic cap on. After ten minutes, check the case for rotation at test points 77, 79, and 81. Is the rotation normal?"
::TRANSLATION	"the case rotation" ! verb("is","is not")! "
END.DEFINE	normal"
DEFINE ATTRIBUTE	change
::DEFINED.ON	imu
::TYPE	text
::LEGAL.VALUES	{sudden.change, gradual.drift}
::LEGAL.MEANS	{query.user}
::DETERMINATION.MEANS	{query.user}
::PROMPT	"How did the RMS get out of spec? Was it a sudden change or a gradual drift?"
::TRANSLATION	"how the RMS got out of spec"
END.DEFINE	
DEFINE ATTRIBUTE	check.a.to.d.converter.not.caged
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"the A-D converter should be checked"
END.DEFINE	
DEFINE ATTRIBUTE	check.a.to.d.converter.servo.disable
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"the A-D converter should be checked"
END.DEFINE	
DEFINE ATTRIBUTE	check.a.to.d.converter.z.stab
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"the A-D converter should be checked"
END.DEFINE	

DEFINE ATTRIBUTE	check.cables
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"the cables may be faulty and should be checked"
END.DEFINE	
DEFINE ATTRIBUTE	check.digital.processor
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"check the digital processor"
END.DEFINE	
DEFINE ATTRIBUTE	check.for.case.rotation
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"the IMU should be checked for case rotation"
END.DEFINE	
DEFINE ATTRIBUTE	check.for.level.platform
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"check to see if the platform is level"
END.DEFINE	
DEFINE ATTRIBUTE	check.gyro.circuit
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"the gyro circuit should be checked"
END.DEFINE	
DEFINE ATTRIBUTE	check.gyro.run.voltage
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"the gyro run voltage should be checked"
END.DEFINE	

DEFINE ATTRIBUTE	check.gyro.torquers
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"check the gyro torquers"
END.DEFINE	
DEFINE ATTRIBUTE	check.meter.m2.position.11
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"the reading on meter m2 position 11 should be checked"
END.DEFINE	
DEFINE ATTRIBUTE	check.meter.m2.position.21
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"the reading on meter m2 position 21 should be checked"
END.DEFINE	
DEFINE ATTRIBUTE	check.pick.off.signals
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"the pick off signals should be checked"
END.DEFINE	
DEFINE ATTRIBUTE	check.pick.off.signals.while.gyros.not.running
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"the pick off signals should be checked while the gyros are not running"
END.DEFINE	
DEFINE ATTRIBUTE	check.position
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"the position should be checked"
END.DEFINE	

```

DEFINE ATTRIBUTE
::DEFINED.ON
::TYPE
::LEGAL.MEANS
::DETERMINATION.MEANS
::TRANSLATION
END.DEFINE

check.power.cube
imu
boolean
{try.rules}
{try.rules}
"check the power cube "

DEFINE ATTRIBUTE
::DEFINED.ON
::TYPE
::LEGAL.MEANS
::DETERMINATION.MEANS
::TRANSLATION
END.DEFINE

check.precision.current.network
imu
boolean
{try.rules}
{try.rules}
"check the precision current network"

DEFINE ATTRIBUTE
::DEFINED.ON
::TYPE
::LEGAL.MEANS
::DETERMINATION.MEANS
::TRANSLATION
END.DEFINE

check.resolver.excitation
imu
boolean
{try.rules}
{try.rules}
"the resolver excitation should be checked"

DEFINE ATTRIBUTE
::DEFINED.ON
::TYPE
::LEGAL.MEANS
::DETERMINATION.MEANS
::TRANSLATION
END.DEFINE

check.resolver.signal.at.imu
imu
boolean
{try.rules}
{try.rules}
"the resolver signal at the IMU should be
checked"

DEFINE ATTRIBUTE
::DEFINED.ON
::TYPE
::LEGAL.MEANS
::DETERMINATION.MEANS
::TRANSLATION
END.DEFINE

check.resolver.signal.at.ncc
imu
boolean
{try.rules}
{try.rules}
"the resolver signal at the NCC should be
checked"

DEFINE ATTRIBUTE
::DEFINED.ON
::TYPE
::LEGAL.MEANS
::DETERMINATION.MEANS
::TRANSLATION
END.DEFINE

check.resolver.signal.on.different.station
imu
boolean
{try.rules}
{try.rules}
"the resolver signal at the IMU should be
checked while the IMU is on a different test
station"

```

DEFINE ATTRIBUTE	check.speed.control.signal
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"the speed control signal should be checked"
END.DEFINE	
DEFINE ATTRIBUTE	check.speed.stability
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"the speed stability should be checked"
END.DEFINE	
DEFINE ATTRIBUTE	check.test.point.13
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"check test point 13"
END.DEFINE	
DEFINE ATTRIBUTE	check.test.point.23
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"check test point 23"
END.DEFINE	
DEFINE ATTRIBUTE	check.velocity.direction
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"the velocity direction should be checked"
END.DEFINE	
DEFINE ATTRIBUTE	check.velocity.meter
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"check the velocity meter"
END.DEFINE	

DEFINE ATTRIBUTE	check.velocity.on.ncc
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"check the velocity on the ncc"
END.DEFINE	
DEFINE ATTRIBUTE	check.vm.signal.on.ncc
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"check the velocity meter signal on the NCC"
END.DEFINE	
DEFINE ATTRIBUTE	contact.te
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"contact the T.E. A fault is suspected in the test equipment"
END.DEFINE	
DEFINE ATTRIBUTE	cooling.hose.is.connected
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{query.user}
::DETERMINATION.MEANS	{query.user}
::PROMPT	"Is the cooling hose connected?"
::TRANSLATION	"the cooling hose " ! verb("is", "is not") ! " connected"
END.DEFINE	
DEFINE ATTRIBUTE	digital.processor.good
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{query.user}
::DETERMINATION.MEANS	{query.user}
::PROMPT	"Replace the digital processor. Is the problem still present?"
::TRANSLATION	"the digital processor " ! verb("is", "is not") ! "good"
END.DEFINE	

DEFINE ATTRIBUTE	exchange.electronic.control.amps
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"the electronic control amps should be exchanged"
END.DEFINE	
DEFINE ATTRIBUTE	exchange.speed.control.cards
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"exchange the speed control cards"
END.DEFINE	
DEFINE ATTRIBUTE	fault.found.continue.testing
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"the fault is identified; continue testing"
END.DEFINE	
DEFINE ATTRIBUTE	faulty.velocity.meter
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{query.user}
::DETERMINATION.MEANS	{query.user}
::PROMPT	"Check the velocity meters. Are they faulty?"
::TRANSLATION	"the velocity meters " ! verb("are", "are not") ! " faulty"
END.DEFINE	
DEFINE ATTRIBUTE	gyro.is.suspect
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"the gyros are suspected"
END.DEFINE	
DEFINE ATTRIBUTE	gyro.run.voltage.good
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{query.user}
::DETERMINATION.MEANS	{query.user}
::PROMPT	"Check the gyro run voltage. Is the voltage good?"
::TRANSLATION	"the gyro run voltage " ! verb("is", "is not") ! " good"
END.DEFINE	

<pre> DEFINE ATTRIBUTE ::DEFINED.ON ::TYPE ::LEGAL.MEANS ::DETERMINATION.MEANS ::PROMPT ::TRANSLATION END.DEFINE </pre>	<pre> gyro.torquers.signals.good imu boolean {query.user} {query.user} "Put the cap on the IMU and check test points 26, 27, and 28. Are the gyro torquers' signals good?" "the gyro torquers " ! verb("are", "are not ") ! "good" </pre>
<pre> DEFINE ATTRIBUTE ::DEFINED.ON ::TYPE ::LEGAL.MEANS ::DETERMINATION.MEANS ::PROMPT ::TRANSLATION END.DEFINE </pre>	<pre> imu.in.caged.mode imu boolean {query.user} {query.user} "Is the IMU in the caged mode?" "the IMU " ! verb("is", "is not") ! "in the caged mode" </pre>
<pre> DEFINE ATTRIBUTE ::DEFINED.ON ::TYPE ::LEGAL.MEANS ::DETERMINATION.MEANS ::PROMPT ::TRANSLATION END.DEFINE </pre>	<pre> imu.resolver.signal.good imu boolean {query.user} {query.user} "Check the resolver signal output at the IMU with the cap on the IMU. Is the signal good?" "the resolver signal output at the IMU " ! verb("is", "is not") ! " good" </pre>
<pre> DEFINE ATTRIBUTE ::DEFINED.ON ::TYPE ::LEGAL.MEANS ::DETERMINATION.MEANS ::PROMPT ::TRANSLATION END.DEFINE </pre>	<pre> imu.resolver.signal.on.different.station.good imu boolean {query.user} {query.user} "Move the IMU to another station. Recheck the excitation to the resolver. Is the signal good?" "the second test of the resolver signal output at a different test station" ! verb("is", "is not") ! " good " </pre>

DEFINE ATTRIBUTE	interchange.electronic.control.amps
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"the electronic control amps should be interchanged"
END.DEFINE	
DEFINE ATTRIBUTE	interchange.gimbal.rate.amps
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"the gimbal rate amps should be interchanged"
END.DEFINE	
DEFINE ATTRIBUTE	interchange.ps1.and.ps2
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"PS1 and PS2 should be interchanged"
END.DEFINE	
DEFINE ATTRIBUTE	interchange.rate.amps
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"The gimbal rate amps should be interchanged"
END.DEFINE	
DEFINE ATTRIBUTE	interchange.speed.control.cards
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"The speed control cards should be interchanged"
END.DEFINE	
DEFINE ATTRIBUTE	monitor.pickoffs
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"monitor the pickoffs"
END.DEFINE	

DEFINE ATTRIBUTE	monitor.speed.control.for.b14.and.up
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"monitor the speed control for B14 and up"
END.DEFINE	
DEFINE ATTRIBUTE	most.drift
::DEFINED.ON	imu
::TYPE	text
::LEGAL.VALUES	{longitude, lattitude}
::LEGAL.MEANS	{query.user}
::DETERMINATION.MEANS	{query.user}
::PROMPT	"What contributes more error to the drift, longitude or lattitude?"
::TRANSLATION	"the main contributor to the drift"
END.DEFINE	
DEFINE ATTRIBUTE	move.to.mode.b
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"the IMU should be moved to Mode B"
END.DEFINE	
DEFINE ATTRIBUTE	new.digital.processor.corrects.problem
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{query.user}
::DETERMINATION.MEANS	{query.user}
::PROMPT	"Replace the digital processor. Did this solve the problem?"
::TRANSLATION	"replacing the digital processor " ! verb("solved", "did not solve") ! "the problem"
END.DEFINE	
DEFINE ATTRIBUTE	next.test.good
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{query.user}
::DETERMINATION.MEANS	{query.user}
::PROMPT	"Run the next test on the Mode A station. Did it pass?"
::TRANSLATION	"the next test " ! verb("did", "did not") ! " pass on the Mode A test station"
END.DEFINE	

DEFINE ATTRIBUTE	normal.response.to.operator.action
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"the error message " ! verb("is", "is not") ! "a normal response to an operator action"
END.DEFINE	
DEFINE ATTRIBUTE	north.south.or.east.west
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{query.user}
::DETERMINATION.MEANS	{query.user}
::PROMPT	"Is the velocity occurring in either the North-South or the East-West direction? "
::TRANSLATION	"the direction of the velocity is either North-South or East-West"
END.DEFINE	
DEFINE ATTRIBUTE	not.indication.of.fault
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"the error message " ! verb("is not", "is") ! "an indication of an IMU fault"
END.DEFINE	
DEFINE ATTRIBUTE	number.of.axes.not.zero
::DEFINED.ON	imu
::TYPE	text
::LEGAL.VALUES	{one, all}
::LEGAL.MEANS	{query.user}
::DETERMINATION.MEANS	{query.user}
::PROMPT	"Check the A to D converter. Is one axis significantly further from zero than the other axes, or are all angles off?"
::TRANSLATION	"number of axes off"
END.DEFINE	

```

DEFINE ATTRIBUTE
::DEFINED.ON
::TYPE
::LEGAL.MEANS
::DETERMINATION.MEANS
::PROMPT

::TRANSLATION

END.DEFINE

DEFINE ATTRIBUTE
::DEFINED.ON
::TYPE
::LEGAL.MEANS
::DETERMINATION.MEANS
::PROMPT

::TRANSLATION

END.DEFINE

DEFINE ATTRIBUTE
::DEFINED.ON
::TYPE
::LEGAL.MEANS
::DETERMINATION.MEANS
::PROMPT
::TRANSLATION

END.DEFINE

DEFINE ATTRIBUTE
::DEFINED.ON
::TYPE
::LEGAL.MEANS
::DETERMINATION.MEANS
::PROMPT

::TRANSLATION

END.DEFINE

```

one.axis.is.further.from.zero
imu
boolean
{query.user}
{query.user}
"Check the A to D converter. Is any axis significantly further from zero than the other axes?"
"one axis " ! verb("is", "is not")!
" significantly further from zero than the rest"

operator.initiated.shutdown
imu
boolean
{query.user}
{query.user}
"Is this message the result of an operator initiated shutdown ?"
"the operator " ! verb("initiated",
"did not initiate") ! "the shutdown"

pick.off.signals.good
imu
boolean
{query.user}
{query.user}
"Check the pick-off signals. Are they good? "
"the pick-off signals " !
verb("are", "are not") ! " good"

pick.off.signals.while.gyros.not.running.good
imu
boolean
{query.user}
{query.user}
"Move the IMU to the Mode B station.
Check the pick-off signals while
the gyros are not running. Are they good?"
"the pick-off signals " ! verb("are", "are
not") ! " good while the gyros are not
running"

DEFINE ATTRIBUTE	position.11.setting.is.normal
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{query.user}
::DETERMINATION.MEANS	{query.user}
::PROMPT	"Check meter M2 in the NCC (setting position 11). Is the reading normal (50%)?"
	"the reading on meter M2 position 11 " !
::TRANSLATION	verb("is", "is not") ! " normal"
END.DEFINE	
DEFINE ATTRIBUTE	position.21.setting.is.normal
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{query.user}
::DETERMINATION.MEANS	{query.user}
::PROMPT	"Check meter M2 in the NCC (setting position 21). Is the reading normal (50%)?"
	"the reading on meter M2 position 21 " !
::TRANSLATION	verb("is", "is not") ! " normal"
END.DEFINE	
DEFINE ATTRIBUTE	power.cube.good
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{query.user}
::DETERMINATION.MEANS	{query.user}
::PROMPT	"Move the IMU to the Mode B station and check the power cube. Is it good?"
::TRANSLATION	"the pover cube " ! verb("is", "is not") ! " good"
END.DEFINE	
DEFINE ATTRIBUTE	power.vas.interrupted
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{query.user}
::DETERMINATION.MEANS	{query.user}
::PROMPT	"Was pover interrupted?"
::TRANSLATION	"pover " ! verb("vas", "vas not") ! " interrupted"
END.DEFINE	
DEFINE ATTRIBUTE	precision.current.network.good
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{query.user}
::DETERMINATION.MEANS	{query.user}
::PROMPT	"Check the precision current network test points 33, 34, and 36. Are the signals good?"
::TRANSLATION	"the precision current network " ! verb("is", "is not") ! "good"
END.DEFINE	

```

DEFINE ATTRIBUTE
::DEFINED.ON
::TYPE
::LEGAL.MEANS
::DETERMINATION.MEANS
::PROMPT
::TRANSLATION
previous.imu.major
imu
boolean
{query.user}
{query.user}
"Has an IMU Major Fault (0504)
occurred previously on this IMU?"
"a previous IMU major fault " ! verb("has",
"has not") ! " occurred on this IMU"

END.DEFINE

DEFINE ATTRIBUTE
::DEFINED.ON
::TYPE
::LEGAL.MEANS
::DETERMINATION.MEANS
::PROMPT
::TRANSLATION
problem.follows.amp
imu
boolean
{query.user}
{query.user}
"Exchange the electronic control amps. Does
the problem follow the card?"
"the problem" ! verb("follows","does not
follow")! " the card"

END.DEFINE

DEFINE ATTRIBUTE
::DEFINED.ON
::TYPE
::LEGAL.MEANS
::DETERMINATION.MEANS
::PROMPT
::TRANSLATION
problem.follows.gimbal.rate.amp
imu
boolean
{query.user}
{query.user}
"Exchange the gimbal rate amps. Does the
problem follow the card?"
"the problem" ! verb("follows","does not
follow")! " the card"

END.DEFINE

DEFINE ATTRIBUTE
::DEFINED.ON
::TYPE
::LEGAL.MEANS
::DETERMINATION.MEANS
::PROMPT
::TRANSLATION
problem.follows.board
imu
boolean
{query.user}
{query.user}
"The servo disable occurred because of excess
rate. Interchange rate amps AR 12, 13, and
14. Does the problem follow the board?"
"the problem" ! verb("follows","does not
follow")! " the card"

END.DEFINE

```

```

DEFINE ATTRIBUTE
::DEFINED.ON
::TYPE
::LEGAL.MEANS
::DETERMINATION.MEANS
::TRANSLATION
END.DEFINE

power.down.message
imu
boolean
{try.rules}
{try.rules}
"the message is a result of a power down"

DEFINE ATTRIBUTE
::DEFINED.ON
::TYPE
::LEGAL.MEANS
::DETERMINATION.MEANS
::PROMPT
::TRANSLATION
END.DEFINE

power.supply.4.8khz.good
imu
boolean
{query.user}
{query.user}
"Install the diagnostic cap and check test
point 23. Is the 4.8 khz power supply good?"
"the 4.8 khz power supply " ! verb("is",
"is not") ! " good"

DEFINE ATTRIBUTE
::DEFINED.ON
::TYPE
::LEGAL.MEANS
::DETERMINATION.MEANS
::PROMPT
::TRANSLATION
END.DEFINE

power.supply.400hz.good
imu
boolean
{query.user}
{query.user}
"Check test point 13. Is the 400 hz power
supply good?"
"the 400 hz power supply " ! verb("is",
"is not") ! " good"

DEFINE ATTRIBUTE
::DEFINED.ON
::TYPE
::LEGAL.MEANS
::DETERMINATION.MEANS
::PROMPT
::TRANSLATION
END.DEFINE

problem.follows.power.supply.card
imu
boolean
{query.user}
{query.user}
"Exchange PS1 and PS2 (the power supply
cards). Does the problem follow the
cards?"
"the problem " ! verb(" follows", "does not
follow") ! " the power supply cards
when PS1 and PS2 are exchanged"

```

DEFINE ATTRIBUTE	problem.follows.speed.control.card
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{query.user}
::DETERMINATION.MEANS	{query.user}
::PROMPT	"Exchange the two speed control cards. Does the problem stay with the same card (ie. did the problem change to the other direction)? "
::TRANSLATION	"the problem " ! verb("follows", "does not follow") ! " the speed control card when the cards are exchanged"
END.DEFINE	
DEFINE ATTRIBUTE	rare.error.message
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"the error message received is rarely received during Mode A tests"
END.DEFINE	
DEFINE ATTRIBUTE	replace.ar8
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"replace.ar8"
END.DEFINE	
DEFINE ATTRIBUTE	replace.digital.processor
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"replace the digital processor"
END.DEFINE	
DEFINE ATTRIBUTE	replace.faulty.component
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"replace the component determined faulty"
END.DEFINE	

DEFINE ATTRIBUTE	replace.platform.electric.switch
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"the platform electric switch should be
replaced"	
END.DEFINE	
DEFINE ATTRIBUTE	replace.4.8.khz.power.supply
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"replace the 4.8 khz power supply"
END.DEFINE	
DEFINE ATTRIBUTE	replacing.ar8.helps
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{query.user}
::DETERMINATION.MEANS	{query.user}
::PROMPT	"Replace AR8 module. Did that solve the
	problem?"
::TRANSLATION	"replacing AR8 " !verb("solved", "did not
	solve") ! " the problem"
END.DEFINE	
DEFINE ATTRIBUTE	replacing.power.supply.solved.problem
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{query.user}
::DETERMINATION.MEANS	{query.user}
::PROMPT	"Replace the 4.8 khz power supply (PS10). Did
	that repair the IMU?"
::TRANSLATION	"replacing the 4.8 khz power supply " !
	verb("repaired","did not repair")!" the IMU"
END.DEFINE	
DEFINE ATTRIBUTE	resolver.excitation.good
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{query.user}
::DETERMINATION.MEANS	{query.user}
::PROMPT	"Check the excitation signal to the
	resolver. Is it good?"
::TRANSLATION	"the excitation signal to the resolver "
	! verb("is", "is not") ! " good"
END.DEFINE	

```

DEFINE ATTRIBUTE
  ::DEFINED.ON
  ::TYPE
  ::LEGAL.MEANS
  ::DETERMINATION.MEANS
  ::PROMPT

  ::TRANSLATION

END.DEFINE

resolver.signal.on.different.station.good
imu
boolean
{query.user}
{query.user}
"Move the IMU to another station. Recheck the
excitation to the resolver. Is the signal
good?"
"the resolver signal " ! verb("is", "is not")
! " good at another test station"

DEFINE ATTRIBUTE
  ::DEFINED.ON
  ::TYPE
  ::LEGAL.MEANS
  ::DETERMINATION.MEANS
  ::PROMPT

  ::TRANSLATION

END.DEFINE

resolver.output.at.ncc.good
imu
boolean
{query.user}
{query.user}
"Check the resolver signal output at
the NCC. Is the signal good?"
"the resolver signal output at the NCC " !
verb("is", "is not") ! " good"

DEFINE ATTRIBUTE
  ::DEFINED.ON
  ::TYPE
  ::LEGAL.MEANS
  ::DETERMINATION.MEANS
  ::PROMPT
  ::TRANSLATION
END.DEFINE

restart.possible
imu
boolean
{query.user}
{query.user}
"Attempt a restart. Is a restart possible?"
"a restart " ! verb("is", "is not") ! " possible"

DEFINE ATTRIBUTE
  ::DEFINED.ON
  ::TYPE
  ::LEGAL.MEANS
  ::DETERMINATION.MEANS
  ::TRANSLATION
END.DEFINE

return.to.test
imu
boolean
{try.rules}
{try.rules}
"return to test after installing the cap"

DEFINE ATTRIBUTE
  ::DEFINED.ON
  ::TYPE
  ::LEGAL.MEANS
  ::DETERMINATION.MEANS
  ::TRANSLATION
END.DEFINE

run.next.test
imu
boolean
{try.rules}
{try.rules}
"the next test " ! verb("was", "was not") ! " run"

```

DEFINE ATTRIBUTE	run.scorsby.test
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"run the scorsby test on the IMU "
END.DEFINE	
DEFINE ATTRIBUTE	scorsby.good
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{query.user}
::DETERMINATION.MEANS	{query.user}
::PROMPT	"Put the IMU on the Scorsby table. Does it pass the Scorsby test?"
::TRANSLATION	"the IMU " ! verb("passed", "did not pass") ! " the Scorsby test"
END.DEFINE	
DEFINE ATTRIBUTE	see.shop.supervisor
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"contact the shop supervisor for assistance in troubleshooting the fault."
END.DEFINE	
DEFINE ATTRIBUTE	speed.control.for.b14.and.up.good
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{query.user}
::DETERMINATION.MEANS	{query.user}
::PROMPT	"Monitor the speed control for B-14 and above. Are the values good?"
::TRANSLATION	"the speed control values for B14 and above " ! verb("are", "are not") ! " good"
END.DEFINE	
DEFINE ATTRIBUTE	speed.control.signal.good
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{query.user}
::DETERMINATION.MEANS	{query.user}
::PROMPT	"Check the speed control signals with the IMU cap on. Are the signals good?"
::TRANSLATION	"the speed control signals " ! verb("are", "are not") ! " good"
END.DEFINE	

DEFINE ATTRIBUTE	suspect.vn
::DEFINED.ON	imu
::TYPE	text
::LEGAL.VALUES	{velocity.meter.x, velocity.meter.y}
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"the suspected velocity meter"
END.DEFINE	
DEFINE ATTRIBUTE	switch.solves.problem
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{query.user}
::DETERMINATION.MEANS	{query.user}
::PROMPT	"Replace the platform electric switch (A7). Does that solve the problem?"
::TRANSLATION	"replacing the platform electric switch " ! verb("did", "did not") ! " solve the problem"
END.DEFINE	
DEFINE ATTRIBUTE	system.achieved.gyro.normal
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{query.user}
::DETERMINATION.MEANS	{query.user}
::PROMPT	"Has the system achieved gyro normal during this test?"
::TRANSLATION	"the system " ! verb("has", "has not") ! "achieved gyro normal during this test"
END.DEFINE	
DEFINE ATTRIBUTE	test
::DEFINED.ON	imu
::TYPE	text
::LEGAL.VALUES	{shim.cal, gyro.cal, self.align, nav.align, navigate}
::LEGAL.MEANS	{query.user}
::DETERMINATION.MEANS	{query.user}
::PROMPT	"What test was the station in when the fault occurred?"
::TRANSLATION	"the current test"
END.DEFINE	

DEFINE ATTRIBUTE	use.accompanying.messages.to.troubleshoot
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"use accompanying messages to troubleshoot the problem"
END.DEFINE	
DEFINE ATTRIBUTE	velocities.changed.directions
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{query.user}
::DETERMINATION.MEANS	{query.user}
::PROMPT	"Continue the cal and wait for the platform to slew. Did the velocities change direction? "
::TRANSLATION	"the velocities changed directions when the platform slewed"
END.DEFINE	
DEFINE ATTRIBUTE	velocity.meter.is.suspect
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{try.rules}
::DETERMINATION.MEANS	{try.rules}
::TRANSLATION	"the velocity meters are suspected"
END.DEFINE	
DEFINE ATTRIBUTE	velocity.on.ncc.is.zero
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{query.user}
::DETERMINATION.MEANS	{query.user}
::PROMPT	"Check the velocity of the velocity meter on the NCC. Is the velocity zero?"
::TRANSLATION	"the velocity of the velocity meter is zero"
END.DEFINE	
DEFINE ATTRIBUTE	vm.signal.on.ncc.good
::DEFINED.ON	imu
::TYPE	boolean
::LEGAL.MEANS	{query.user}
::DETERMINATION.MEANS	{query.user}
::PROMPT	"Check the velocity meter signal on the NCC. Is it good?"
::TRANSLATION	"the velocity meter signal " ! verb("is", "is not") ! "good on the NCC"
END.DEFINE	

```

DEFINE ATTRIBUTE                                wait.for.platform.to.slew
::DEFINED.ON                                  imu
::TYPE                                        boolean
::LEGAL.MEANS                                {try.rules}
::DETERMINATION.MEANS                        {try.rules}
::TRANSLATION                                "wait for the platform to slew"
END.DEFINE

DEFINE ATTRIBUTE                                yz.fault.occurred.more.than.twice
::DEFINED.ON                                  imu
::TYPE                                        boolean
::LEGAL.MEANS                                {query.user}
::DETERMINATION.MEANS                        {query.user}
::PROMPT                                     "Has the YZ gyro speed control fault
                                             occurred more than twice?"
::TRANSLATION                                "the YZ gyro speed control fault " !
                                             verb("has", "has not") ! " occurred
                                             more than twice"

END.DEFINE

DEFINE ATTRIBUTE                                yz.speed.control.stable
::DEFINED.ON                                  imu
::TYPE                                        boolean
::LEGAL.MEANS                                {query.user}
::DETERMINATION.MEANS                        {query.user}
::PROMPT                                     "Move the IMU to the Mode B station.
                                             Is the YZ speed control stable?"
::TRANSLATION                                "the YZ speed control "
                                             ! verb("is", "is not") ! " stable"

END.DEFINE

```

```

/*****
/*
/*                               VALUE HIERARCHIES
/*
/* These are value hierarchy definitions
/* Error Messages list all 62 possible error messages that can be
/* generated by the DMINS Automatic Test Equipment.
*/

```

```

DEFINE VALUE.HIERARCHY error.messages
::SUBVALUES      {velocity.unreasonable,
                  vt.greater.than.2.knots,
                  xy.speed.control,
                  yz.speed.control,
                  imu.major,
                  no.input.3.axes,
                  no.input.az,
                  no.input.pitch,
                  no.input.roll,
                  automatic.shutdown,
                  imu.o.load,
                  imu.o.temp,
                  pwr.interrupt,
                  dcc.o.load.o.temp,
                  dcc.o.temp,
                  i.c.fault,
                  comp.tie.in.sv.on,
                  i.c.fault.inhb.enab,
                  seq.cnt.no.compare,
                  i.c.data.loop.fault,
                  i.c.fault.cont,
                  in.parity.test.inhb.enab,
                  out.word.par.inhb.enab,
                  input.parity.fault,
                  output.word.parity.fault,
                  output.word.parity.cont,
                  excess.angle,
                  servo.disable,
                  major.reset.fault,
                  z.stab,
                  system.in.free.run,
                  minor.reset.fault,
                  minor.fault.cont,
                  imu.minor,
                  plat.stab.abort,
                  xvm.precounter.fault,
                  yvm.precounter.fault,
                  both.vvm.precounter.failure,
                  vvm.bite.failure,
                  x.gyro.torque.fault,
                  y.gyro.torque.fault,
                  z.gyro.torque.fault,
                  system.not.properly.caged,
                  gyro.hot,

```

```

gyro.cold,
gyro.temp.normal,
mux.decoder.dl.fault,
cage.xy.dl.fault,
cage.yz.dl.fault,
gyro.start.dl.fault,
gyro.run.dl.fault,
uyk.good.dl.fault,
input.parity.dl.fault.mux04,
input.parity.no.2.dl.fault,
output.word.parity.dl.fault.mux09,
vt.vr.greater.than.3.knots,
minisins.vel.dif.exceeds.limit,
minisins.pos.dif.exceeds.limit,
parity.test.1.no.go,
parity.test.2.no.go,
parity.test.3.no.go,
put.intercom.test.no.go,
rms.out.of.spec)
::TRANSLATION " error messages "
::VALUE " error.messages "
END.DEFINE

/* Possible Faults include the 38 Shop Replaceable Units (SRU) in a */
/* DMINS IMU as well as common terms for groups of components (such as */
/* "speed cards") often used by DMINS technicians. */

DEFINE VALUE.HIERARCHY possible.faults
::SUBVALUES {bandpass.filter.and.shift.register.1,
bandpass.filter.and.shift.register.2,
precision.torquing.driver.x,
precision.torquing.driver.y,
precision.torquing.driver.z,
platform.electronic.switch,
shorting.plug,
precision.current.network,
stable.platform,
displacement.gyroscope.xy,
displacement.gyroscope.yz,
velocity.meter.x,
velocity.meter.y,
resolver.buffer.amp,
gyro.buffer.amp.xy,
gyro.buffer.amp.yz,
xy.640hz.power.supply,
yz.640hz.power.supply,
power.cube,
no.1.400hz.power.supply,
no.2.400hz.power.supply,
triangle.generator.and.case.rotation.power.supply,
power.supply.4.8khz,
frequency.standard,
d.c.amplifier.xy,

```

d.c.amplifier.yz,
 synchro.signal.buffer.amp,
 gyro.cage.amp,
 thermoelectric.signal.amp,
 gyro.temperature.controller,
 gimbal.cage.amp,
 platform.signal.amp,
 platform.electronic.control.amp.roll,
 platform.electronic.control.amp.pitch,
 platform.electronic.control.amp.azimuth,
 gimbal.rate.electronic.control.amp.roll,
 gimbal.rate.electronic.control.amp.pitch,
 gimbal.rate.electronic.control.amp.azimuth,
 corresponding.gimbal.rate.electronic.control.amp,
 corresponding.gimbal.rate.amp,
 corresponding.electronic.control.amp,
 platform.electric.switch,
 corresponding.gyro,
 gyro,
 corresponding.velocity.meter,
 speed.card,
 corresponding.speed.card,
 power.supply.card,
 corresponding.power.supply.card,
 resolver,
 excitation.module,
 bad.cable,
 none,
 not.determined)

::TRANSLATION
 ::VALUE
 END.DEFINE

" possible faults "
 "possible.faults"

```

/*****
/*
/*                      RULES                      */
/*
/* Rules in this program are in groups according to the error */
/* message they are attempting to troubleshoot. This is done to */
/* increase the maintainability of the program.                */
/*
/* The first group of rules define actions that are concluded */
/* throughout the program. They are placed at the beginning of the */
/* program to increase readability.                            */
/*
/* The attribute "action" is used to pass recommendations to the user. */
/* "Fault" is determined so that S.1 can be queried for faults of */
/* previous sessions using the "what" command. If no fault can be */
/* determined, fault defaults to not.determined.              */
*/

```

```

DEFINE RULE rule001
::APPLIED.TO I:IMU
::PREMISE    not.indication.of.fault[I]
::CONCLUSION begin
              action[I] = "continue testing. This error message is
                          not an indication of a fault in the IMU";
              fault[I] = none;
              end

```

END.DEFINE

```

DEFINE RULE rule002
::APPLIED.TO I:IMU
::PREMISE    power.down.message[I]
::CONCLUSION begin
              action[I] = "This error message is not an indication of
                          an IMU fault. Usually, it is associated with
                          a powerdown. If this is not the case,
                          contact T.E. to inspect the test equipment";
              fault[I] = none;
              end

```

END.DEFINE

```

DEFINE RULE rule003
::APPLIED.TO I:IMU
::PREMISE    use.accompanying.messages.to.troubleshoot[I]
::CONCLUSION action[I] = "This error message is normally seen with other
                          messages. It does not contribute any useful
                          information. Use the error messages that
                          accompany this message for troubleshooting"

```

END.DEFINE

```

DEFINE RULE rule004
::APPLIED.TO I:IMU
::PREMISE    replace.faulty.component[I]
::CONCLUSION action[I] = "Recommend that the "!fault[I] !" be replaced"
END.DEFINE

```

```

DEFINE RULE rule005
::APPLIED.TO I:IMU
::PREMISE    contact.te[I]
::CONCLUSION begin
               action[I] = "Call T.E. to check the test equipment. There
               is reason to believe the test equipment is at fault";
               fault[I] = none;
            end
END.DEFINE

DEFINE RULE rule006
::APPLIED.TO I:IMU
::PREMISE    see.shop.supervisor[I]
::CONCLUSION action[I]="this is a difficult problem to troubleshoot.
Contact the shop supervisor for further assistance. Once
the fault has been determined, contact the engineering
section so this program can be updated to include the new
information"
END.DEFINE

DEFINE RULE rule007
::APPLIED.TO I:IMU
::PREMISE    move.to.mode.b[I]
::CONCLUSION action[I] ="Move the IMU to Mode B for further diagnostics"
END.DEFINE

DEFINE RULE rule008
::APPLIED.TO I:IMU
::PREMISE    fault.found.continue.testing[I]
::CONCLUSION action[I] = "Continue testing. The fault was in the " !
               fault[I]
END.DEFINE

DEFINE RULE rule009
::APPLIED.TO I:IMU
::PREMISE    rare.error.message[I]
::CONCLUSION action[I] = "This error message is not common during Mode A
tests. Contact the shop supervisor for further
assistance. Once the fault has been
determined, contact the engineering section so
this program can be updated to include the new
information"
END.DEFINE

```

```

/*****
/* This group of rules check the error message to determine if there */
/* is an actual fault in the IMU or if the error message was due to a */
/* routine action. This group includes rules 009 - 012 and corresponds */
/* to the following error messages: */
/*
/*          input.parity.fault, */
/*          output.word.parity.fault, */
/*          output.word.parity.cont, */
/*          gyro.temp.normal, */
/*          mux.decoder.dl.fault, */
/*          cage.xy.dl.fault, */
/*          cage.yz.dl.fault, */
/*          gyro.start.dl.fault, */
/*          gyro.run.dl.fault, */
/*          uyk.good.dl.fault, */
/*          input.parity.dl.fault.mux04, */
/*          input.parity.no.2.dl.fault, */
/*          output.word.parity.dl.fault.mux09, */
/*          imu.minor. */

```

DEFINE RULE rule010

::APPLIED.TO I:IMU

::PREMISE error.message[I] is in {input.parity.fault,
output.word.parity.fault,
output.word.parity.cont,
gyro.temp.normal}

::CONCLUSION not.indication.of.fault[I]

END.DEFINE

DEFINE RULE rule011

::APPLIED.TO I:IMU

::PREMISE error.message[I] is in {mux.decoder.dl.fault,
cage.xy.dl.fault,
cage.yz.dl.fault,
gyro.start.dl.fault,
gyro.run.dl.fault,
uyk.good.dl.fault,
input.parity.dl.fault.mux04,
input.parity.no.2.dl.fault,
output.word.parity.dl.fault.mux09}

::CONCLUSION power.down.message[I]

END.DEFINE

DEFINE RULE rule012

::APPLIED.TO I:IMU

::PREMISE error.message[I] is imu.minor

::CONCLUSION use.accompanying.messages.to.troubleshoot[I]

END.DEFINE

DEFINE RULE rule016

::APPLIED.TO I:IMU

::PREMISE check.resolver.signal.at.ncc[I] and
not resolver.output.at.ncc.good[I]

::CONCLUSION check.resolver.signal.at.imu[I]

END.DEFINE

DEFINE RULE rule017

::APPLIED.TO I:IMU

::PREMISE check.resolver.signal.at.imu[I] and
imu.resolver.signal.good[I]

::CONCLUSION check.resolver.signal.on.different.station[I]

END.DEFINE

DEFINE RULE rule018

::APPLIED.TO I:IMU

::PREMISE check.resolver.signal.at.imu[I] and
not imu.resolver.signal.good[I]

::CONCLUSION check.resolver.excitation[I]

END.DEFINE

DEFINE RULE rule019

::APPLIED.TO I:IMU

::PREMISE check.resolver.signal.on.different.station[I] and
resolver.signal.on.different.station.good[I]

::CONCLUSION contact.te[I]

END.DEFINE

DEFINE RULE rule020

::APPLIED.TO I:IMU

::PREMISE check.resolver.signal.on.different.station[I] and
not resolver.signal.on.different.station.good[I]

::CONCLUSION check.resolver.excitation[I]

END.DEFINE

DEFINE RULE rule021

::APPLIED.TO I:IMU

::PREMISE check.resolver.excitation[I] and
resolver.excitation.good[I]

::CONCLUSION begin
replace.faulty.component[I];
fault[I] = resolver;
end

END.DEFINE

DEFINE RULE rule022

```
::APPLIED.TO I:IMU
::PREMISE    check.resolver.excitation[I] and
              not resolver.excitation.good[I]
::CONCLUSION begin
              replace.faulty.component[I];
              fault[I] = excitation.module;
              end
```

END.DEFINE

```
/*****
/*          ***      Rules for Transition      ***
*/
/* Transition from shallow to deep reasoning occurs in Rules 24,25,35, */
/* 36,38, and 41. Rules 26,27,28,29,39,40, and 42 were eliminated */
/* by commenting them out. The eliminated rules were left in place to */
/* provide an example of the procedure for future additions to the */
/* program. */
/* Rules 023 - 029 determines the cause of the following messages: */
/*                               xy.speed.control, */
/*                               yz.speed.control. */
```

DEFINE RULE rule023

```
::APPLIED.TO I:IMU
::PREMISE    error.message[I] is yz.speed.control and
              not yz.fault.occurred.more.than.twice[I]
::CONCLUSION begin
              action[I] = "No action is required. If the YZ speed
                           control error has not occurred more than
                           twice, it is normally not a problem";
              fault[I] = none;
              end
```

END.DEFINE

DEFINE RULE rule024

```
::APPLIED.TO I:IMU
::PREMISE    error.message[I] is xy.speed.control
::CONCLUSION action[I] = "Begin deep diagnosis"
```

END.DEFINE

DEFINE RULE rule025

```
::APPLIED.TO I:IMU
::PREMISE    error.message[I] is yz.speed.control and
              yz.fault.occurred.more.than.twice[I]
::CONCLUSION action[I] = "Begin deep diagnosis"
```

END.DEFINE

/***** ELIMINATED *****/
Rules 26, 27, 28, and 29 were eliminated and replaced by model diagnosis

DEFINE RULE rule026

::APPLIED.TO I:IMU

::PREMISE check.speed.control.signal[I] and
speed.control.signal.good[I]

::CONCLUSION action[I] = "Check the discretes on the modules. A
module problem is suspected"

END.DEFINE

DEFINE RULE rule027

::APPLIED.TO I:IMU

::PREMISE check.speed.control.signal[I] and
not speed.control.signal.good[I]

::CONCLUSION exchange.speed.control.cards[I]

END.DEFINE

DEFINE RULE rule028

::APPLIED.TO I:IMU

::PREMISE exchange.speed.control.cards[I] and
not problem.follows.speed.control.card[I]

::CONCLUSION begin
replace.faulty.component[I];
fault[I] = corresponding.gyro;
end

END.DEFINE

DEFINE RULE rule029

::APPLIED.TO I:IMU

::PREMISE exchange.speed.control.cards[I] and
problem.follows.speed.control.card[I]

::CONCLUSION begin
replace.faulty.component[I];
fault[I] = corresponding.speed.card;
end

END.DEFINE

```

*****/
/*****/
/* This group of rules includes rules 030 - 042 and troubleshoots the */
/* following two error messages: */
/*                                velocity.unreasonable, */
/*                                vt.greater.than.2.knots. */
/* In addition, this section uses the routine Check Gyro Circuit, which */
/* starts with rule 043. */

```

```

DEFINE RULE rule030
::APPLIED.TO I:IMU
::PREMISE    error.message[I] is in {velocity.unreasonable,
                                vt.greater.than.2.knots} and
                                test[I] is gyro.cal
::CONCLUSION check.position[I]
END.DEFINE

```

```

DEFINE RULE rule031
::APPLIED.TO I:IMU
::PREMISE    error.message[I] is in {velocity.unreasonable,
                                vt.greater.than.2.knots} and
                                test[I] is navigate
::CONCLUSION check.velocity.direction[I]
END.DEFINE

```

```

DEFINE RULE rule032
::APPLIED.TO I:IMU
::PREMISE    error.message[I] is in {velocity.unreasonable,
                                vt.greater.than.2.knots} and
                                not test[I] is in {gyro.cal, navigate}
::CONCLUSION action[I]= "See the shop supervisor. Normally this message
                                occurs only when the IMU is in gyro.cal or
                                navigate"
END.DEFINE

```

```

DEFINE RULE rule033
::APPLIED.TO I:IMU
::PREMISE    check.position[I] and
                                north.south.or.east.vest[I]
::CONCLUSION wait.for.platform.to.slew[I]
END.DEFINE

```

```

DEFINE RULE rule034
::APPLIED.TO I:IMU
::PREMISE    wait.for.platform.to.slew[I] and
                                velocities.changed.directions[I]
::CONCLUSION begin
                                replace.faulty.component[I];
                                fault[I] = corresponding.velocity.meter;
                                end
END.DEFINE

```

```

DEFINE RULE rule035
::APPLIED.TO I:IMU
::PREMISE    wait.for.platform.to.slew[I] and
              not velocities.changed.directions[I]
::CONCLUSION action[I] ="Begin deep diagnosis"
END.DEFINE

DEFINE RULE rule036
::APPLIED.TO I:IMU
::PREMISE    check.position[I] and
              not north.south.or.east.vest[I]
::CONCLUSION action[I] = "Begin deep diagnosis"
END.DEFINE

DEFINE RULE rule037
::APPLIED.TO I:IMU
::PREMISE    check.velocity.direction[I] and
              north.south.or.east.vest[I]
::CONCLUSION begin
              replace.faulty.component[I];
              fault[I] = corresponding.velocity.meter;
              end
END.DEFINE

DEFINE RULE rule038
::APPLIED.TO I:IMU
::PREMISE    check.velocity.direction[I] and
              not north.south.or.east.vest[I]
::CONCLUSION action[I] = "Begin deep diagnosis"
END.DEFINE

/***** ELIMINATED *****/
Rules 39 and 40 were eliminated and replaced by model based diagnosis.

DEFINE RULE rule039
::APPLIED.TO I:IMU
::PREMISE    check.for.level.platform[I] and
              not azimuth.equal.zero[I]
::CONCLUSION check.velocity.meter[I]
END.DEFINE

DEFINE RULE rule040
::APPLIED.TO I:IMU
::PREMISE    check.velocity.meter[I] and
              faulty.velocity.meter[I]
::CONCLUSION begin
              replace.faulty.component[I];
              fault[I] = corresponding.velocity.meter;
              end
END.DEFINE
*****/

```

```

DEFINE RULE rule041
::APPLIED.TO I:IMU
::PREMISE    check.for.level.platform[I] and
              azimuth.equal.zero[I]
::CONCLUSION action[I] = "Begin deep diagnosis"
END.DEFINE

```

```

/***** ELIMINATED *****/
Rule 42 was eliminated and replaced by model based diagnosis.

```

```

DEFINE RULE rule042
::APPLIED.TO I:IMU
::PREMISE    check.velocity.meter[I] and
              not faulty.velocity.meter[I]
::CONCLUSION check.gyro.circuit[I]
END.DEFINE

```

```

*****/
/*****/
/*
/*          CHECK GYRO CIRCUIT ROUTINE          */
/* This routine is used for several different error messages including */
/* velocity unreasonable, vt. greater than 2 knots, imu major, and    */
/* plat stab abort. Note: This routine is no longer used for velocity  */
/* unreasonable or vt greater than 2 knots. The deep reasoning portion */
/* of the program now provides the logic for these messages.          */
*/

```

```

DEFINE RULE rule043
::APPLIED.TO I:IMU
::PREMISE    check.gyro.circuit[I] and
              speed.control.signal.good[I]
::CONCLUSION check.gyro.run.voltage[I]
END.DEFINE

```

```

DEFINE RULE rule044
::APPLIED.TO I:IMU
::PREMISE    check.gyro.run.voltage[I] and
              gyro.run.voltage.good[I]
::CONCLUSION check.pick.off.signals[I]
END.DEFINE

```

```

DEFINE RULE rule045
::APPLIED.TO I:IMU
::PREMISE    check.pick.off.signals[I] and
              pick.off.signals.good[I]
::CONCLUSION check.pick.off.signals.while.gyros.not.running[I]
END.DEFINE

```

DEFINE RULE rule046

```
::APPLIED.TO I:IMU
::PREMISE    check.pick.off.signals[I] and
              not pick.off.signals.good[I]
::CONCLUSION begin
              replace.faulty.component[I];
              fault[I] = corresponding.gyro;
              end
```

END.DEFINE

DEFINE RULE rule047

```
::APPLIED.TO I:IMU
::PREMISE    check.pick.off.signals.while.gyros.not.running[I] and
              not pick.off.signals.while.gyros.not.running.good[I]
::CONCLUSION begin
              replace.faulty.component[I];
              fault[I] = corresponding.gyro;
              end
```

END.DEFINE

DEFINE RULE rule048

```
::APPLIED.TO I:IMU
::PREMISE    check.pick.off.signals.while.gyros.not.running[I] and
              pick.off.signals.while.gyros.not.running.good[I]
::CONCLUSION monitor.speed.control.for.b14.and.up[I]
```

END.DEFINE

DEFINE RULE rule049

```
::APPLIED.TO I:IMU
::PREMISE    monitor.speed.control.for.b14.and.up[I] and
              speed.control.for.b14.and.up.good[I]
::CONCLUSION run.next.test[I]
```

END.DEFINE

DEFINE RULE rule050

```
::APPLIED.TO I:IMU
::PREMISE    monitor.speed.control.for.b14.and.up[I] and
              not speed.control.for.b14.and.up.good[I]
::CONCLUSION begin
              replace.faulty.component[I];
              fault[I] = corresponding.gyro;
              end
```

END.DEFINE

DEFINE RULE rule051

```
::APPLIED.TO I:IMU
::PREMISE    check.gyro.run.voltage[I] and
              not gyro.run.voltage.good[I]
::CONCLUSION interchange.ps1.and.ps2[I]
```

END.DEFINE

```

DEFINE RULE rule052
::APPLIED.TO I:IMU
::PREMISE interchange.ps1.and.ps2[I] and
problem.follows.power.supply.card[I]
::CONCLUSION begin
replace.faulty.component[I];
fault[I] = corresponding.power.supply.card;
end
END.DEFINE

```

```

DEFINE RULE rule053
::APPLIED.TO I:IMU
::PREMISE interchange.ps1.and.ps2[I] and
not problem.follows.power.supply.card[I]
::CONCLUSION begin
replace.faulty.component[I];
fault[I] = corresponding.gyro;
end
END.DEFINE

```

```

DEFINE RULE rule054
::APPLIED.TO I:IMU
::PREMISE check.gyro.circuit[I] and
not speed.control.signal.good[I]
::CONCLUSION interchange.speed.control.cards[I]
END.DEFINE

```

```

DEFINE RULE rule055
::APPLIED.TO I:IMU
::PREMISE interchange.speed.control.cards[I] and
not problem.follows.speed.control.card[I]
::CONCLUSION begin
replace.faulty.component[I];
fault[I] = corresponding.gyro;
end
END.DEFINE

```

```

DEFINE RULE rule056
::APPLIED.TO I:IMU
::PREMISE interchange.speed.control.cards[I] and
problem.follows.speed.control.card[I]
::CONCLUSION begin
replace.faulty.component[I];
fault[I] = corresponding.speed.card;
end
END.DEFINE

```

```

DEFINE RULE rule057
::APPLIED.TO I:IMU
::PREMISE run.next.test[I] and
not next.test.good[I]
::CONCLUSION check.speed.stability[I]
END.DEFINE

```

```

DEFINE RULE rule058
::APPLIED.TO I:IMU
::PREMISE      run.next.test[I] and
                next.test.good[I]
::CONCLUSION   run.scorsby.test[I]
END.DEFINE

DEFINE RULE rule059
::APPLIED.TO I:IMU
::PREMISE      run.scorsby.test[I] and
                not scorsby.good[I]
::CONCLUSION   check.speed.stability[I]
END.DEFINE

DEFINE RULE rule060
::APPLIED.TO I:IMU
::PREMISE      run.scorsby.test[I] and
                scorsby.good[I]
::CONCLUSION   begin
                action[I] = "Testing is complete. All inputs indicate
                            this testing is good";
                fault[I] = none;
                end
END.DEFINE

/*****
/* This group of rules includes rules 061 - 064 and corresponds to an */
/* imu major error message. This section also uses the Check Gyro */
/* Circuit Routine which starts with rule 043. */

DEFINE RULE rule061
::APPLIED.TO I:IMU
::PREMISE      error.message[I] is imu.major and
                not previous.imu.major[I]
::CONCLUSION   check.gyro.circuit[I]
END.DEFINE

DEFINE RULE rule062
::APPLIED.TO I:IMU
::PREMISE      error.message[I] is imu.major and
                previous.imu.major[I]
::CONCLUSION   check.speed.stability[I]
END.DEFINE

DEFINE RULE rule063
::APPLIED.TO I:IMU
::PREMISE      check.speed.stability[I] and
                yz.speed.control.stable[I]
::CONCLUSION   begin
                replace.faulty.component[I];
                fault[I] = displacement.gyroscope.xy;
                end
END.DEFINE

```

```

DEFINE RULE rule064
::APPLIED.TO I:IMU
::PREMISE    check.speed.stability[I] and
              not yz.speed.control.stable[I]
::CONCLUSION begin
              replace.faulty.component[I];
              fault[I] = displacement.gyroscope.yz;
            end
END.DEFINE

/*****
/* This group of rules includes rules 065 - 071 and corresponds to the */
/* following error messages:                                           */
/*                               automatic.shutdown,                  */
/*                               pwr.interrupt.                        */
/*                               */
/*****/

DEFINE RULE rule065
::APPLIED.TO I:IMU
::PREMISE    error.message[I] is in {automatic.shutdown,
                                      pwr.interrupt} and
              operator.initiated.shutdown[I]
::CONCLUSION normal.response.to.operator.action[I]
END.DEFINE

DEFINE RULE rule066
::APPLIED.TO I:IMU
::PREMISE    normal.response.to.operator.action[I]
::CONCLUSION begin
              action[I]="The error message received is a normal
                          response to an operator action. No action
                          is necessary" ;
              fault[I] = none;
            end
END.DEFINE

DEFINE RULE rule067
::APPLIED.TO I:IMU
::PREMISE    error.message[I] is in {automatic.shutdown,
                                      pwr.interrupt} and
              not operator.initiated.shutdown[I]
::CONCLUSION attempt.restart[I]
END.DEFINE

DEFINE RULE rule068
::APPLIED.TO I:IMU
::PREMISE    attempt.restart[I] and
              not restart.possible[I]
::CONCLUSION check.power.cube[I]
END.DEFINE

```



```

/*****
/* This group of rules includes rules 075 - 076 and corresponds to the */
/* system in free run error message. */

```

DEFINE RULE rule075

```

::APPLIED.TO I:IMU
::PREMISE    error.message[I] is system.in.free.run and
              power.was.interrupted[I]
::CONCLUSION action[I] = "restore power. If problem persists,
                      contact the shop supervisor"

```

END.DEFINE

DEFINE RULE rule076

```

::APPLIED.TO I:IMU
::PREMISE    error.message[I] is system.in.free.run and
              not power.was.interrupted[I]
::CONCLUSION see.shop.supervisor[I]

```

END.DEFINE

```

/*****
/* This group of rules includes rule 077 and corresponds to plat stab */
/* abort. It also uses the Check Gyro Circuit Routine which starts */
/* with rule 043 */

```

DEFINE RULE rule077

```

::APPLIED.TO I:IMU
::PREMISE    error.message[I] is plat.stab.abort
::CONCLUSION check.gyro.circuit[I]

```

END.DEFINE

```

/*****
/* This group of rules includes rules 078 - 085 and corresponds to the */
/* following error messages: */
/* */
/* */
/* */
/* */

```

DEFINE RULE rule078

```

::APPLIED.TO I:IMU
::PREMISE    error.message[I] is xvm.precounter.fault
::CONCLUSION begin
              check.velocity.on.ncc[I];
              suspect.vm[I]=velocity.meter.x ;
              end

```

END.DEFINE

DEFINE RULE rule079

```

::APPLIED.TO I:IMU
::PREMISE    error.message[I] is yvm.precounter.fault
::CONCLUSION begin
              check.velocity.on.ncc[I]; suspect.vm[I]=velocity.meter.y ;
              end

```

END.DEFINE

```

DEFINE RULE rule080
::APPLIED.TO I:IMU
::PREMISE    check.velocity.on.ncc[I] and
              velocity.on.ncc.is.zero[I]
::CONCLUSION check.digital.processor[I]
END.DEFINE

DEFINE RULE rule081
::APPLIED.TO I:IMU
::PREMISE    check.digital.processor[I] and
              digital.processor.good[I]
::CONCLUSION contact.te[I]
END.DEFINE

DEFINE RULE rule082
::APPLIED.TO I:IMU
::PREMISE    check.digital.processor[I] and
              not digital.processor.good[I]
::CONCLUSION begin
              action[I] = "Continue testing. The fault was in the
                           digital processor";
              fault[I] = none;
              end
END.DEFINE

DEFINE RULE rule083
::APPLIED.TO I:IMU
::PREMISE    check.velocity.on.ncc[I] and
              not velocity.on.ncc.is.zero[I]
::CONCLUSION check.vm.signal.on.ncc[I]
END.DEFINE

DEFINE RULE rule084
::APPLIED.TO I:IMU
::PREMISE    check.vm.signal.on.ncc[I] and
              vm.signal.on.ncc.good[I]
::CONCLUSION move.to.mode.b[I]
END.DEFINE

DEFINE RULE rule085
::APPLIED.TO I:IMU
::PREMISE    check.vm.signal.on.ncc[I] and
              not vm.signal.on.ncc.good[I]
::CONCLUSION begin
              replace.faulty.component[I];
              fault[I] = suspect.vm[I];
              end
END.DEFINE

```

```

/*****
/* This group of rules includes rules 086 - 092 and corresponds to the */
/* following error messages:                                         */
/*           x.gyro.torque.fault,                                   */
/*           y.gyro.torque.fault,                                   */
/*           z.gyro.torque.fault                                     */

```

DEFINE RULE rule086

::APPLIED.TO I:IMU

::PREMISE error.message[I] is in {x.gyro.torque.fault,
y.gyro.torque.fault,
z.gyro.torque.fault}

::CONCLUSION replace.digital.processor[I]

END.DEFINE

DEFINE RULE rule087

::APPLIED.TO I:IMU

::PREMISE replace.digital.processor[I] and
new.digital.processor.corrects.problem[I]

::CONCLUSION begin
action[I] = "Continue testing. The fault was in the
digital processor";
fault[I] = none;
end

END.DEFINE

DEFINE RULE rule088

::APPLIED.TO I:IMU

::PREMISE replace.digital.processor[I] and
not new.digital.processor.corrects.problem[I]

::CONCLUSION check.gyro.torquers[I]

END.DEFINE

DEFINE RULE rule089

::APPLIED.TO I:IMU

::PREMISE check.gyro.torquers[I] and
gyro.torquers.signals.good[I]

::CONCLUSION see.shop.supervisor[I]

END.DEFINE

DEFINE RULE rule090

::APPLIED.TO I:IMU

::PREMISE check.gyro.torquers[I] and
not gyro.torquers.signals.good[I]

::CONCLUSION check.precision.current.network[I]

END.DEFINE

```

DEFINE RULE rule091
::APPLIED.TO I:IMU
::PREMISE      check.precision.current.network[I] and
                 not precision.current.network.good[I]
::CONCLUSION   begin
                 replace.faulty.component[I];
                 fault[I] = precision.current.network;
                 end
END.DEFINE

DEFINE RULE rule092
::APPLIED.TO I:IMU
::PREMISE      check.precision.current.network[I] and
                 precision.current.network.good[I]
::CONCLUSION   see.shop.supervisor[I]
END.DEFINE

/*****
/* This group of rules includes rules 093 - 094 and corresponds to the */
/* gyro cold error message.                                           */
*****/

DEFINE RULE rule093
::APPLIED.TO I:IMU
::PREMISE      error.message[I] is gyro.cold and
                 system.achieved.gyro.normal[I]
::CONCLUSION   move.to.mode.b[I]
END.DEFINE

DEFINE RULE rule094
::APPLIED.TO I:IMU
::PREMISE      error.message[I] is gyro.cold and
                 not system.achieved.gyro.normal[I]
::CONCLUSION   begin
                 action[I] = "No action is required. This message is
                               common during the first 20 to 30 minutes of
                               testing while the gyro warms up" ;
                 fault[I] = none;
                 end
END.DEFINE

/*****
/* This group of rules includes rules 095 - 100 and corresponds to the */
/* gyro hot error message.                                           */
*****/

DEFINE RULE rule095
::APPLIED.TO I:IMU
::PREMISE      error.message[I] is gyro.hot and
                 not cooling.hose.is.connected[I]
::CONCLUSION   begin
                 action[I] = "reconnect the cooling hose to the IMU";
                 fault[I] = none;
                 end
END.DEFINE

```

```

DEFINE RULE rule096
::APPLIED.TO I:IMU
::PREMISE      error.message[I] is gyro.hot and
                  cooling.hose.is.connected[I]
::CONCLUSION    check.meter.m2.position.21[I]
END.DEFINE

DEFINE RULE rule097
::APPLIED.TO I:IMU
::PREMISE      check.meter.m2.position.21[I] and
                  not position.21.setting.is.normal[I]
::CONCLUSION    move.to.mode.b[I]
END.DEFINE

DEFINE RULE rule098
::APPLIED.TO I:IMU
::PREMISE      check.meter.m2.position.21[I] and
                  position.21.setting.is.normal[I]
::CONCLUSION    check.meter.m2.position.11[I]
END.DEFINE

DEFINE RULE rule099
::APPLIED.TO I:IMU
::PREMISE      check.meter.m2.position.11[I] and
                  not position.11.setting.is.normal[I]
::CONCLUSION    action[I] = "return to Mode B station and perform test B22.
                           The high start circuit is faulty"
END.DEFINE

DEFINE RULE rule100
::APPLIED.TO I:IMU
::PREMISE      check.meter.m2.position.11[I] and
                  position.11.setting.is.normal[I]
::CONCLUSION    action[I] = "return to Mode B station and perform test B22.
                           The gyro and associated circuitry is
                           suspected"
END.DEFINE

/*****
/* This group of rules includes rules 101 - 107 and corresponds to */
/* system not properly caged.                                     */

DEFINE RULE rule101
::APPLIED.TO I:IMU
::PREMISE      error.message[I] is system.not.properly.caged
::CONCLUSION    check.a.to.d.converter.not.caged[I]
END.DEFINE

```

```

DEFINE RULE rule102
::APPLIED.TO I:IMU
::PREMISE      check.a.to.d.converter.not.caged[I] and
                 one.axis.is.further.from.zero[I]
::CONCLUSION    replace.platform.electric.switch[I]
END.DEFINE

DEFINE RULE rule103
::APPLIED.TO I:IMU
::PREMISE      check.a.to.d.converter.not.caged[I] and
                 not one.axis.is.further.from.zero[I]
::CONCLUSION    move.to.mode.b[I]
END.DEFINE

DEFINE RULE rule104
::APPLIED.TO I:IMU
::PREMISE      replace.platform.electric.switch[I] and
                 switch.solves.problem[I]
::CONCLUSION    begin
                 fault.found.continue.testing[I];
                 fault[I] = platform.electric.switch;
                 end
END.DEFINE

DEFINE RULE rule105
::APPLIED.TO I:IMU
::PREMISE      replace.platform.electric.switch[I] and
                 not switch.solves.problem[I]
::CONCLUSION    interchange.electronic.control.amps[I]
END.DEFINE

DEFINE RULE rule106
::APPLIED.TO I:IMU
::PREMISE      interchange.electronic.control.amps[I] and
                 problem.follows.amp[I]
::CONCLUSION    begin
                 replace.faulty.component[I];
                 fault[I] = corresponding.electronic.control.amp;
                 end
END.DEFINE

DEFINE RULE rule107
::APPLIED.TO I:IMU
::PREMISE      interchange.electronic.control.amps[I] and
                 not problem.follows.amp[I]
::CONCLUSION    begin
                 replace.faulty.component[I];
                 fault[I] = gimbal.cage.amp;
                 end
END.DEFINE

```

/* This group of rules includes rules 108 -116 and corresponds to zstab */

DEFINE RULE rule108

::APPLIED.TO I:IMU

::PREMISE error.message[I] is z.stab

::CONCLUSION check.a.to.d.converter.z.stab[I]

END.DEFINE

DEFINE RULE rule109

::APPLIED.TO I:IMU

::PREMISE check.a.to.d.converter.z.stab[I] and
number.of.axes.not.zero[I] is all

::CONCLUSION check.for.case.rotation[I]

END.DEFINE

DEFINE RULE rule110

::APPLIED.TO I:IMU

::PREMISE check.for.case.rotation[I] and
case.rotation.normal[I]

::CONCLUSION move.to.mode.b[I]

END.DEFINE

DEFINE RULE rule111

::APPLIED.TO I:IMU

::PREMISE check.for.case.rotation[I] and
not case.rotation.normal[I]

::CONCLUSION begin
replace.faulty.component[I]; fault[I] =
corresponding.gyro;
end

END.DEFINE

DEFINE RULE rule112

::APPLIED.TO I:IMU

::PREMISE check.a.to.d.converter.z.stab[I] and
number.of.axes.not.zero[I] is one

::CONCLUSION interchange.gimbal.rate.amps[I]

END.DEFINE

DEFINE RULE rule113

::APPLIED.TO I:IMU

::PREMISE interchange.gimbal.rate.amps[I] and
problem.follows.gimbal.rate.amp[I]

::CONCLUSION begin
replace.faulty.component[I];
fault[I] = corresponding.gimbal.rate.amp;
end

END.DEFINE

```

DEFINE RULE rule114
::APPLIED.TO I:IMU
::PREMISE interchange.gimbal.rate.amps[I] and
not problem.follows.gimbal.rate.amp[I]
::CONCLUSION exchange.electronic.control.amps[I]
END.DEFINE

DEFINE RULE rule115
::APPLIED.TO I:IMU
::PREMISE exchange.electronic.control.amps[I] and
problem.follows.amp[I]
::CONCLUSION begin
replace.faulty.component[I];
fault[I] = corresponding.electronic.control.amp;
end
END.DEFINE

DEFINE RULE rule116
::APPLIED.TO I:IMU
::PREMISE exchange.electronic.control.amps[I] and
not problem.follows.amp[I]
::CONCLUSION move.to.mode.b[I]
END.DEFINE

/*****
/* This group of rules includes rules 117 - 127 and corresponds to */
/* servo disable. */

DEFINE RULE rule117
::APPLIED.TO I:IMU
::PREMISE error.message[I] is servo.disable
::CONCLUSION check.a.to.d.converter.servo.disable[I]
END.DEFINE

DEFINE RULE rule118
::APPLIED.TO I:IMU
::PREMISE check.a.to.d.converter.servo.disable[I] and
all.three.axes.near.zero[I]
::CONCLUSION check.test.point.23[I]
END.DEFINE

DEFINE RULE rule119
::APPLIED.TO I:IMU
::PREMISE check.a.to.d.converter.servo.disable[I] and
not all.three.axes.near.zero[I]
::CONCLUSION interchange.rate.amps[I]
END.DEFINE

```

```

DEFINE RULE rule120
::APPLIED.TO I:IMU
::PREMISE interchange.rate.amps[I] and
not problem.follows.board[I]
::CONCLUSION move.to.mode.b[I]
END.DEFINE

DEFINE RULE rule121
::APPLIED.TO I:IMU
::PREMISE interchange.rate.amps[I] and
problem.follows.board[I]
::CONCLUSION begin
replace.faulty.component[I];
fault[I]=corresponding.gimbal.rate.electronic.control.amp;
end
END.DEFINE

DEFINE RULE rule122
::APPLIED.TO I:IMU
::PREMISE check.test.point.23[I] and
power.supply.4.8khz.good[I]
::CONCLUSION check.test.point.13[I]
END.DEFINE

DEFINE RULE rule123
::APPLIED.TO I:IMU
::PREMISE check.test.point.23[I] and
not power.supply.4.8khz.good[I]
::CONCLUSION replace.4.8.khz.power.supply[I]
END.DEFINE

DEFINE RULE rule124
::APPLIED.TO I:IMU
::PREMISE replace.4.8.khz.power.supply[I] and
replacing.power.supply.solved.problem[I]
::CONCLUSION begin
fault.found.continue.testing[I];
fault[I] = power.supply.4.8khz;
end
END.DEFINE

DEFINE RULE rule125
::APPLIED.TO I:IMU
::PREMISE replace.4.8.khz.power.supply[I] and
not replacing.power.supply.solved.problem[I]
::CONCLUSION move.to.mode.b[I]
END.DEFINE

```

```

DEFINE RULE rule126
::APPLIED.TO I:IMU
::PREMISE      check.test.point.13[I] and
                power.supply.400hz.good[I]
::CONCLUSION    action[I]="This appears to be a intermittent problem.
                    Recommend the IMU be returned to Mode B and the
                    slip rings be inspected"
END.DEFINE

DEFINE RULE rule127
::APPLIED.TO I:IMU
::PREMISE      check.test.point.13[I] and
                not power.supply.400hz.good[I]
::CONCLUSION    action[I]="Replace or troubleshoot the 400hz boards"
END.DEFINE

/*****
/* This group of rules includes rules 126 - 134 and corresponds to */
/* excess angle */
DEFINE RULE rule128
::APPLIED.TO I:IMU
::PREMISE      error.message[I] is excess.angle and not
                imu.in.caged.mode[I]
::CONCLUSION    not.indication.of.fault[I]
END.DEFINE

DEFINE RULE rule129
::APPLIED.TO I:IMU
::PREMISE      error.message[I] is excess.angle and
                imu.in.caged.mode[I]
::CONCLUSION    return.to.test[I]
END.DEFINE

DEFINE RULE rule130
::APPLIED.TO I:IMU
::PREMISE      return.to.test[I] and
                not able.to.restart[I]
::CONCLUSION    move.to.mode.b[I]
END.DEFINE

DEFINE RULE rule131
::APPLIED.TO I:IMU
::PREMISE      return.to.test[I] and
                able.to.restart[I]
::CONCLUSION    monitor.pickoffs[I]
END.DEFINE

DEFINE RULE rule132
::APPLIED.TO I:IMU
::PREMISE      monitor.pickoffs[I] and
                angle.occurs.first.on[I] is xy.pickoff
::CONCLUSION    replace.ar8[I]
END.DEFINE

```

```

DEFINE RULE rule133
  ::APPLIED.TO I:IMU
  ::PREMISE      monitor.pickoffs[I] and
                  angle.occurs.first.on[I] is yz.pickoff
  ::CONCLUSION   replace.ar8[I]
END.DEFINE

DEFINE RULE rule134
  ::APPLIED.TO I:IMU
  ::PREMISE      replace.ar8[I] and
                  replacing.ar8.helps[I]
  ::CONCLUSION   begin
                  fault.found.continue.testing[I];
                  fault[I] = platform.signal.amp;
                  end
END.DEFINE

DEFINE RULE rule135
  ::APPLIED.TO I:IMU
  ::PREMISE      replace.ar8[I] and
                  not replacing.ar8.helps[I] and
                  angle.occurs.first.on[I] is xy.pickoff
  ::CONCLUSION   action[I]= "Return to Mode B station. The XY gyro may be
                           faulty"
END.DEFINE

DEFINE RULE rule136
  ::APPLIED.TO I:IMU
  ::PREMISE      replace.ar8[I] and
                  not replacing.ar8.helps[I] and
                  angle.occurs.first.on[I] is yz.pickoff
  ::CONCLUSION   action[I]= "Return to Mode B station. The YZ gyro may be
                           faulty"
END.DEFINE

/*****
/* This group of rules includes rules 137 - 142 and corresponds to RMS */
/* out of spec. This is not an error message, but is an important    */
/* condition that requires diagnosis.                                   */
/

DEFINE RULE rule137
  ::APPLIED.TO I:IMU
  ::PREMISE      error.message[I] is rms.out.of.spec and
                  change[I] is sudden.change
  ::CONCLUSION   velocity.meter.is.suspect[I]
END.DEFINE

```

```

DEFINE RULE rule138
::APPLIED.TO I:IMU
::PREMISE      error.message[I] is rms.out.of.spec and
                change[I] is gradual.drift
::CONCLUSION   gyro.is.suspect[I]
END.DEFINE

DEFINE RULE rule139
::APPLIED.TO I:IMU
::PREMISE      velocity.meter.is.suspect[I] and
                most.drift[I] is longitude
::CONCLUSION   action[I] = "The Y velocity meter is suspected as the cause
                        of the problem"
END.DEFINE

DEFINE RULE rule140
::APPLIED.TO I:IMU
::PREMISE      velocity.meter.is.suspect[I] and
                most.drift[I] is latitude
::CONCLUSION   action[I] = "The X velocity meter is suspected as the cause
                        of the problem"
END.DEFINE

DEFINE RULE rule141
::APPLIED.TO I:IMU
::PREMISE      gyro.is.suspect[I] and
                most.drift[I] is longitude
::CONCLUSION   action[I] = "The XY gyro is suspected as the cause of the
                        problem"
END.DEFINE

DEFINE RULE rule142
::APPLIED.TO I:IMU
::PREMISE      gyro.is.suspect[I] and
                most.drift[I] is latitude
::CONCLUSION   action[I] = "The YZ gyro is suspected as the cause of the
                        problem"
END.DEFINE

```

```

/*****
/*****
/*
/*          Part II: Deep Reasoning Diagnostics          */
/*          1Lt Jim Skinner                               */
/*          21 Aug 88                                     */
/*
/* This portion of the program uses a model-based approach to diagnose */
/* faults in the Inertial Measurement Unit (IMU) of the Dual Miniature */
/* Inertial Navigation System. Only a subset of the IMU is diagnosed  */
/* in this program. This subset includes the velocity meter function,  */
/* the gyro speed control function, and the gyro torquing function.    */
/*
/*****

```

```

DEFINE CONTROL.BLOCK          DIAGNOSE.IMU.FAULT
::INVOCATION                  internal
::ARGUMENTS                   I:imu
::TRANSLATION                  "determine the fault in the"!
                                "IMU"

::BODY begin vars d:dummy,
    high:path,
    med:path,
    low:path;
    create.instance dummy called d with
    name.of[d] = "nul";
    create.instance path called high with
    search.level[high] = high.level;
    create.instance path called med with
    search.level[med] = med.level;
    create.instance path called low with
    search.level[low] = low.level;
    determine error.message[I];
    if error.message[I] is in {velocity.unreasonable,
                                vt.greater.than.2.knots,
                                xy.speed.control,
                                yz.speed.control}
    then invoke deep.diagnose(I,I,d,high)
    else display nev.line() ! "This error message has not been"!
                                "included in the model."
end;
END.DEFINE

```

```

/*****
/*                                DEEP DIAGNOSE                                */
/* This Control Block is the Inference Engine of this portion of the */
/* program. Although it is based on a program from the S.1 manual, it */
/* is greatly enhanced. The control block determines the fault of the */
/* system by searching for a bad input to the current system. If one is */
/* found, the control block is called with the component responsible for*/
/* the bad input as its argument. This continues until a component is */
/* found with a bad output, but no bad inputs. It is then determined if */
/* this component has subcomponents. If so, the subcomponents are built */
/* and the subcomponents are diagnosed. When a component is found with */
/* a bad output, good inputs, and no subcomponents, the component is */
/* identified as the faulty component and the search is terminated. The */
/* algorithm has been enhanced to ensure highest risk components (the */
/* components most likely to be at fault) are tested first followed */
/* by components with medium risk, then those with low risk. In the */
/* case of components with multiple inputs, the high risk inputs will */
/* be tested first. This is due to the ordering of the Rules that cause */
/* the components to be tested. The Arguments for this control block */
/* are:
/* Component - The component currently under consideration.
/* Start - The first component in the current path being tested.
/* Finish - The last component in the current path being tested.
/*          (Initially, a dummy variable since the end is not known)
/* P:Path - The level of the current search. This dictates whether a
/*          component will be tested immediately. Initially, the
/*          search level of the path will be set to high level and
/*          only components with high risk or multiple inputs will be
/*          tested. If the fault is not found, the search level will
/*          be lowered to medium level, then finally low level.
DEFINE CONTROL.BLOCK                                DEEP.DIAGNOSE
::INVOCATION                                internal
::ARGUMENTS                                COMPONENT:imu.component,
                                           START:imu.component,
                                           FINISH:imu.component,
                                           p:path

::BODY
begin
display new.line()!
new.line()! "Deep Diagnose was invoked with"!
new.line()! "component = "! instance.trans(component)!
new.line()! "start = "! instance.trans(start)!
new.line()! "finish = "!instance.trans(finish)!
new.line()! "level = "! search.level[p];

```

Case search.level[p] of
high.level:

/* In a high level search end will always equal 0 because as soon as an end point is found, level of search will be changed to medium. Also, the output of start is always bad. A critical test component is defined as one that is high risk or has multiple inputs. The pseudo code for high level is:

if not critical.test(component) and not end.point(component)
then call deep diagnose(component1,start,0,high)

if not critical.test(component) and end.point(component)
then call deep.diagnose(start,start,component,med)

if critical.test(component) and output.test.ok(component)
then call deep.diagnose(start,start,component,med)

if critical.test(component) and not output.test.ok(component)
then determine bad.input[component]

if bad.input exists call
deep.diagnose(component1,component1,0,high)
if bad.input does not exist determine if component
has subcomponents. If so, call
deep.diagnose(component2,component2,0,high)
if component does not have subcomponents create
instance fault with name.of[component]. */

begin

determine critical.test[COMPONENT];

if not critical.test[COMPONENT]

then begin

if end.point[COMPONENT]

then begin

if exists(med:path|search.level[med] is
med.level)

then invoke deep.diagnose(start,start,component,med)

end

else begin

if exists(COMPONENT1:imu.component,

high:path|connected.to[COMPONENT1,COMPONENT] known
and search.level[high] is high.level)

then invoke deep.diagnose(component1,start,finish,high)

end

end

else begin

determine output.test.ok[component];

if output.test.ok[component]

then begin

if exists(med:path|search.level[med] is med.level)

then invoke deep.diagnose(start,start,component,med)

end

```

else begin
    determine bad.input[COMPONENT];
    if bad.input[COMPONENT] definite
    then begin
        if exists(COMPONENT1:imu.component,high:path|
            connected.to[COMPONENT1,COMPONENT] =
            bad.input[COMPONENT] and
            search.level[high] is high.level)
        then invoke deep.diagnose(component1,component1,finish,high)
        end
    else begin
        if has.subcomponent[COMPONENT]
        then begin
            invoke build.subcomponent(COMPONENT);
            if exists (COMPONENT2:imu.component,
                high:path,d:dummy|
                same.output.as[COMPONENT,COMPONENT2]
                and search.level[high] is high.level)
            then begin
                invoke deep.diagnose(COMPONENT2,COMPONENT2,d,high);
            end
        end
        else display new.line()! "The fault was found"!
            " to be the " ! instance.name(COMPONENT);
        end;
    end;
end;

med.level:

/* There are two ways med is called from the high level.
First, if the end.point is found so that the fault has been
isolated between either start and end.point or a high and an
endpoint. Second if it has been isolated between two
endpoints. Either way, a start and an end are known. Also,
the output of start is known to be bad. End is never 0. There
will be no high risk or multiple input components in the
path. */

/* if risk(component) is low and not component = end then
call deep diagnose(component1,start,end,med) */

/* if risk(component) is low and component = end then
call deep diagnose(start,start,end,low) */

/* if risk(component) is med and output.test.ok(component)
then call deep.diagnose(start,start,component,low) */

/* if risk(component) is med and not output.test.ok(component)
then determine bad.input[component] */

/* if bad.input exists call

```

```

    deep.diagnose(component1,component1,0,med)*/

/* if bad.input does not exist determine if component
has subcomponents. If so, call
deep.diagnose(component2,component2,0,high) */

/* if component does not have subcomponents create
instance fault with name.of[component]. */

begin
  if risk[component] is low
  then begin
    if finish = component
    then begin
      if exists(low:path|search.level[low] is low.level)
      then invoke deep.diagnose(start,start,finish,low)
      end
    else begin
      if exists(COMPONENT1:imu.component,med:path|
        connected.to[COMPONENT1,COMPONENT] known and
        search.level[med] is med.level)
      then invoke deep.diagnose(component1,start,finish,med)
      end
    end
  else if output.test.ok[COMPONENT]
  then begin
    if exists(low:path|search.level[low] is low.level)
    then invoke deep.diagnose(start,start,component,low)
    end
  else begin
    determine bad.input[COMPONENT];
    if bad.input[COMPONENT] definite
    then begin
      if exists(COMPONENT1:imu.component,d:dummy,
        med:path|connected.to[COMPONENT1,COMPONENT] =
        bad.input[COMPONENT] and search.level[med]
        is med.level)
      then invoke deep.diagnose(component1,component1,d,med)
      end
    else begin
      if has.subcomponent[COMPONENT]
      then begin
        invoke build.subcomponent(COMPONENT);
        if exists (COMPONENT2:imu.component,
          d:dummy,high:path|
          same.output.as[COMPONENT,COMPONENT2]
          and search.level[high] is high.level)
        then invoke
          deep.diagnose(COMPONENT2,COMPONENT2,d,high)
        end
      else display new.line()! "The fault was found"!
        " to be the " ! instance.name(COMPONENT);
      end
    end
  end
end

```

```

        end

end;

lov.level:
/* risk is known to be low.*/
/* input known to be single.*/
/* the output is bad */

/* if bad.input exists call
   deep.diagnose(component1,component1,0,low) */

/* if bad.input does not exist determine if component
   has subcomponents. If so, call
   deep.diagnose(component2,component2,0,high) */

/* if component does not have subcomponents create
   instance fault with name.of[component]. */

begin
  determine bad.input[COMPONENT];
  if bad.input[COMPONENT] definite
  then begin
    if exists(COMPONENT1:imu.component,low:path|
      connected.to[COMPONENT1,COMPONENT] =
      bad.input[COMPONENT] and search.level[low] is
      lov.level)
    then invoke deep.diagnose(component1,component1,finish,low)
    end
  else begin
    if has.subcomponent[COMPONENT]
    then begin
      invoke build.subcomponent(COMPONENT);
      if exists (COMPONENT2:imu.component,d:dummy,
        high:path|same.output.as[COMPONENT,COMPONENT2]
        and search.level[high] is high.level)
      then invoke deep.diagnose(COMPONENT2,COMPONENT2,d,high)
      end
    else display nev.line()! "The fault was found"!
      " to be the " ! instance.name(COMPONENT);
    end
  end
end

end;
end;
END.DEFINE

```

```

/*****
/* This control block directs the program to the correct control block */
/* for constructing the subcomponents of a component for further      */
/* diagnosis. */

```

```

DEFINE CONTROL.BLOCK          BUILD.SUBCOMPONENT
::INVOCATION                  internal
::ARGUMENTS                   COMPONENT:imu.component
::TRANSLATION                 "build the logical subcomponent"!
                               "of the system"
::BODY

begin
  vars I:imu,
    GTF:gyro.torque.function,
    GSCF0:gyro.speed.control.function.output,
    GSCFE:gyro.speed.control.function.error,
    VMF:velocity.meter.function;

  case name.of[COMPONENT] of
    "IMU": begin
      if exists I:IMU
        then invoke build.imu(I);
      end;
    "gyro torquing": begin
      if exists GTF:gyro.torque.function
        then invoke build.gyro.torque.function(GTF);
      end;
    "gyro speed": begin
      if exists
        GSCF0:gyro.speed.control.function.output
      then invoke build.speed.control.output(GSCF0);
      end;
    "speed error": begin if exists
      GSCFE:gyro.speed.control.function.error
    then invoke build.speed.error(GSCFE);
    end;
    "velocity": begin if exists VMF:velocity.meter.function
      then invoke build.velocity.function(VMF);
    end
      end;
    end;
  end;
END.DEFINE

```

/*****/

```
DEFINE CONTROL.BLOCK          BUILD.IMU
::INVOCATION                  internal
::ARGUMENTS                   I:IMU
::TRANSLATION                  "build the logical subcomponent"!
                                "of the main system"

::BODY
begin
  vars  EMO:error.message.output,
        STPT:sequence.timing.function.torque,
        STPR:sequence.timing.function.run,
        PTF:platform.torquing.function,
        GTF:gyro.torque.function,
        GSCFO:gyro.speed.control.function.output,
        GSCFE:gyro.speed.control.function.error,
        VMF:velocity.meter.function;

  display new.line()! new.line()! "The subcomponents of the IMU"!
    " will be constructed.";

  create.instance error.message.output called EMO with
  begin
    same.output.as[I,EMO];
    risk[EMO] = low;
    multiple.input[EMO];
  end;

  create.instance velocity.meter.function called VMF with
  begin
    has.subcomponent[VMF];
    name.of[VMF] = "velocity";
    connected.to[VMF,EMO] = 1;
    risk[VMF] = med;
  end;

  create.instance platform.torquing.function called PTF with
  begin
    name.of[PTF] = "platform";
    connected.to[PTF,VMF] = 1;
    risk[PTF] = low;
  end;

  create.instance gyro.speed.control.function.output called
  GSCFO with
  begin
    has.subcomponent[GSCFO];
    name.of[GSCFO] = "gyro speed";
    connected.to[GSCFO,PTF] = 1;
    risk[GSCFO] = high; multiple.input[GSCFO]; end;

  create.instance gyro.speed.control.function.error called GSCFE
  with
```

```

begin
  has.subcomponent[GSCFE];
  name.of[GSCFE] = "speed error";
  connected.to[GSCFE,EMO] = 2;
  risk[GSCFE] = high;
  multiple.input[GSCFE];
end;

create.instance gyro.torque.function called GTF with
begin
  has.subcomponent[GTF];
  name.of[GTF] = "gyro torquing";
  connected.to[GTF,GSCFE] = 2;
  connected.to[GTF,GSCFO] = 2;
  risk[GTF] = high;
end;

create.instance sequence.timing.function.run called STFR with
begin
  name.of[STFR] = "sequence timing run";
  connected.to[STFR,GSCFE] = 1;
  connected.to[STFR,GSCFO] = 1;
  risk[STFR] = low;
  end.point[STFR];
end;

create.instance sequence.timing.function.torque called STFT with
begin
  name.of[STFT] = "sequence timing torque";
  connected.to[STFT,GTF] = 1;
  risk[STFT] = low;
  end.point[STFT];
end;

end;
END.DRFINE

```

/*****/

DEFINE CONTROL.BLOCK

::INVOCATION

::ARGUMENTS

::TRANSLATION

::BODY

begin

vars T.E:test.equipment,
X.VM:x.velocity.meter,
Y.VM:y.velocity.meter,
P:power.distribution,
F:frequency.standard;

display new.line()! new.line()! "The fault has been isolated"!
" to the velocity meter function. The subcomponents of the"!
" velocity meter function will be constructed.";

create.instance test.equipment called T.E with

begin

same.output.as[VMF,T.E];
risk[T.E] = low;
multiple.input[T.E];

end;

if exists PTF:platform.torquing.function

then create.instance x.velocity.meter called X.VM with

begin

connected.to[X.VM,T.E] = 1;
connected.to[PTF,X.VM] = 1;
risk[X.VM] = high;
multiple.input[X.VM];

end;

if exists PTF:platform.torquing.function

then create.instance y.velocity.meter called Y.VM with

begin

connected.to[Y.VM,T.E] = 2;
connected.to[PTF,Y.VM] = 1;
risk[Y.VM] = high;
multiple.input[Y.VM];

end;

create.instance frequency.standard called F with

begin

connected.to[F,X.VM] = 2;
connected.to[F,Y.VM] = 2;
risk[F] = low;
end.point[F];

end;

BUILD.VELOCITY.FUNCTION

internal

VMF:velocity.meter.function

"build the logical subcomponent"

"of the velocity meter function"

```
create.instance power.distribution called P with
begin
  connected.to[P,X.VM] = 3;
  connected.to[P,Y.VM] = 3;
  risk[P] = low;
  end.point[P];
end;
end;
END.DEFINE
```

```

/*****/
DEFINE CONTROL.BLOCK          BUILD.GYRO.TORQUE.FUNCTION
::INVOCATION                  internal
::ARGUMENTS                   GTF:gyro.torque.function
::TRANSLATION                  "build the logical subcomponent"!
                               "of the gyro torque function"

::BODY
begin
  vars GT:gyro.torquer,
        TD:torque.driver,
        PCN:precision.current.network,
        GTFR:gyro.torquer.field.return;

  display new.line()! new.line()! "The fault has been isolated"!
  " to the gyro torque function. The subcomponents of the"!
  " gyro torque function will be constructed.";

  create.instance gyro.torquer called GT with
  begin
    same.output.as[GTF,GT];
    risk[GT] = high;
  end;

  if exists STFT:sequence.timing.function.torque
  then create.instance torque.driver called TD with
  begin
    connected.to[TD,GT] = 1;
    connected.to[STFT,TD] = 2;
    risk[TD] = med;
    multiple.input[TD];
  end;

  create.instance precision.current.network called PCN with
  begin
    connected.to[PCN,TD] = 1;
    risk[PCN] = med;
  end;

  create.instance gyro.torquer.field.return called GTFR with
  begin
    connected.to[GTFR,PCN] = 1;
    risk[GTFR] = high;
    end.point[GTFR];
  end;
end;

END.DEFINE

```

```

/*****
/* This control block is used if an xy or yz speed control error is */
/* received. The case where the Gyro Speed Control Function is */
/* identified as faulty by tracing the signal back through the Platform */
/* Torquing Function is handled in the control block labeled Build */
/* Speed Control Output. */

```

```

DEFINE CONTROL.BLOCK

```

```

::INVOCATION

```

```

::ARGUMENTS

```

```

::TRANSLATION

```

```

BUILD.SPEED.ERROR

```

```

internal

```

```

GSF:gyro.speed.control.function.error

```

```

"build the logical subcomponent"!

```

```

"of the gyro speed control function"

```

```

::BODY

```

```

begin

```

```

vars DC:dc.amp,
      BPF:band.pass.filter,
      GB:gyro.buffer.ECA,
      GX:xy.gyroscope,
      GY:yz.gyroscope,
      SR:slip.ring,
      PS:power.supply,
      DCRC:d.c.run.control;

```

```

display new.line()! new.line()! "The fault has been isolated"!
" to the gyro speed control function. The subcomponents of the"!
" gyro speed control function will be constructed.";

```

```

if exists STFR:sequence.timing.function.run
then create.instance dc.amp called DC with
begin
  same.output.as[GSF,DC];
  connected.to[STFR,DC] = 2;
  risk[DC] = low;
  multiple.input[DC];
end;

```

```

create.instance band.pass.filter called BPF with
begin
  connected.to[BPF,DC] = 1;
  risk[BPF] = med;
end;

```

```

create.instance gyro.buffer.eca called GB with
begin
  connected.to[GB,BPF] = 1;
  risk[GB] = med;
end;

```

```

if exists(I:imu| error.message[I] is xy.speed.control)
then if exists GTF:gyro.torque.function
  then create.instance xy.gyroscope called GX with
  begin
    connected.to[GX,GB] = 1;

```

```

        connected.to[GTF,GX] = 2;
        risk[GX] = high;
        multiple.input[GX];
    end;

    if exists(I:imu| error.message[I] is xy.speed.control)
    then create.instance slip.ring called SR with
        begin
            connected.to[SR,GX] = 1;
            risk[SR] = low;
        end;

    if exists(I:imu| error.message[I] is yz.speed.control)
    then if exists GTF:gyro.torque.function
        then create.instance yz.gyroscope called GY with
            begin
                connected.to[GY,GB] = 1;
                connected.to[GTF,GY] = 2;
                risk[GY] = high;
                multiple.input[GY];
            end;

    if exists(I:imu| error.message[I] is yz.speed.control)
    then create.instance slip.ring called SR with
        begin
            connected.to[SR,GY] = 1;
            risk[SR] = low;
        end;

    create.instance power.supply called PS with
        begin
            connected.to[PS,SR] = 1;
            risk[PS] = low;
        end;

    create.instance d.c.run.control called DCRC with
        begin
            connected.to[DCRC,PS] = 1;
            risk[DCRC] = low;
            end.point[DCRC];
        end;

    end;

END.DEFINE

```

```

/*****
/* This control block handles constructing the subcomponents of the */
/* Gyro Speed Control Function in the event the fault has been traced */
/* from the Platform Torquing Function. */

```

```

DEFINE CONTROL.BLOCK          BUILD.SPEED.CONTROL.OUTPUT
::INVOCATION                  internal
::ARGUMENTS                   GSF:gyro.speed.control.function.output
::TRANSLATION                  "build the logical subcomponent"!
                              "of the gyro speed control function"

```

```

::BODY
begin vars DC:dc.amp,
          BPF:band.pass.filter,
          GB:gyro.buffer.ECA,
          GX:xy.gyroscope,
          GY:yz.gyroscope,
          G:gyroscope,
          SR:slip.ring,
          PS:power.supply,
          DCRC:d.c.run.control;

```

```

display new.line()! new.line()! "The fault has been isolated"!
" to the gyro speed control function. The subcomponents of the"!
" gyro speed control function will be constructed.";

```

```

determine bad.pickoff[GSF];
if bad.pickoff[GSF] is XY
then begin
    if exists GTF:gyro.torque.function
    then create.instance xy.gyroscope called GX with
        begin
            same.output.as[GSF,GX];
            connected.to[GTF,GX] = 2;
            risk[GX] = high;
            multiple.input[GX];
        end;
    create.instance slip.ring called SR with
        begin
            connected.to[SR,GX] = 1;
            risk[SR] = low;
        end;
    end;

```

```

if bad.pickoff[GSF] is YZ
then begin
    if exists GTF:gyro.torque.function
    then create.instance yz.gyroscope called GY with
        begin
            same.output.as[GSF,GY];
            connected.to[GTF,GY] = 2;
            risk[GY] = high;
            multiple.input[GY];
        end;

```

```

        create.instance slip.ring called SR with
        begin
            connected.to[SR,GY] = 1;
            risk[SR] = low;
        end;
    end;

    if bad.pickoff[GSP] not.known
    .then begin
        if exists GTF:gyro.torque.function then create.instance
        gyroscope called G with
        begin
            same.output.as[GSP,G];
            connected.to[GTF,G] = 2;
            risk[G] = high;
            multiple.input[G];
        end;
        create.instance slip.ring called SR with
        begin
            connected.to[SR,G] = 1;
            risk[SR] = low;
        end; end;

    create.instance power.supply called PS with
    begin
        connected.to[PS,SR] = 1;
        risk[PS] = low;
    end;

    if exists STFR:sequence.timing.function.run
    then create.instance d.c.run.control called DCRC with
    begin
        connected.to[DCRC,PS] = 1;
        connected.to[STFR,DCRC] = 2;
        risk[DCRC] = low;
        multiple.input[DCRC];
    end;

    create.instance band.pass.filter called BPF with
    begin
        connected.to[BPF,DCRC] = 1;
        risk[BPF] = med;
    end;

    create.instance gyro.buffer.eca called GB with
    begin
        connected.to[GB,BPF] = 1;
        risk[GB] = med;
        end.point[GB];
    end;
end;
END.DEFINE

```

```

/*****
/*                                CLASS TYPE                                */
/* This program contains a single class type, IMU.COMPONENT. All of the */
/* classes are of 's0type.                                           */

DEFINE CLASS.TYPE          IMU.COMPONENT
::CONTAINS                {imu,
                           sequence.timing.function.run,
                           sequence.timing.function.torque,
                           platform.torquing.function,
                           gyro.torque.function,
                           gyro.speed.control.function.output,
                           gyro.speed.control.function.error,
                           velocity.meter.function,
                           error.message.output,
                           test.equipment,
                           x.velocity.meter,
                           y.velocity.meter,
                           power.distribution,
                           frequency.standard,
                           gyro.torquer,
                           gyro.torquer.field.return,
                           torque.driver,
                           precision.current.network,
                           gyro.torquer.field.return,
                           dc.amp,
                           band.pass.filter,
                           gyro.buffer.eca,
                           xy.gyroscope,
                           yz.gyroscope,
                           gyroscope,
                           slip.ring,
                           dummy,
                           power.supply,
                           d.c.run.control}

::CLASS.TYPE.TRANSLATION  "an IMU component"
::PLURAL.CLASS.TYPE.TRANSLATION "the IMU components"
::BLAND.INSTANCE.TRANSLATION "the IMU component"
END.DEFINE

```

```

/*****
/*
/*          CLASSES
/*
/* There are 29 classes, one for each function or component of the IMU.
/* The classes are listed here in alphabetical order.
*/
*/

DEFINE CLASS band.pass.filter
::NUMBER.INSTANCE 1
::ANNOUNCEMENT new.line()! "A model of the "!
               "band pass filter was created."
::CLASS.TRANSLATION "the band pass filter"
::FULL.INSTANCE.TRANSLATION "the band pass filter"
::PLURAL.CLASS.TRANSLATION "the band pass filter"
::BLAND.INSTANCE.TRANSLATION "the band pass filter"
END.DEFINE

DEFINE CLASS dc.amp
::NUMBER.INSTANCE 1
::ANNOUNCEMENT new.line()! "A model of the DC amp"!
               " was created."
::CLASS.TRANSLATION "the DC amp"
::FULL.INSTANCE.TRANSLATION "the DC amp"
::PLURAL.CLASS.TRANSLATION "the DC amp"
::BLAND.INSTANCE.TRANSLATION "the DC amp"
END.DEFINE

DEFINE CLASS d.c.run.control
::NUMBER.INSTANCE 1
::ANNOUNCEMENT new.line()! "A model of the DC"!
               " run control was created."
::CLASS.TRANSLATION "the DC run control"
::FULL.INSTANCE.TRANSLATION "the DC run control"
::PLURAL.CLASS.TRANSLATION "the DC run control"
::BLAND.INSTANCE.TRANSLATION "the DC run control"
END.DEFINE

DEFINE CLASS dummy
::NUMBER.INSTANCE 1
::CLASS.TRANSLATION "a dummy component"
::FULL.INSTANCE.TRANSLATION "a dummy component"
END.DEFINE

DEFINE CLASS error.message.output
::NUMBER.INSTANCE 1
::CLASS.TRANSLATION "the error message output"
::FULL.INSTANCE.TRANSLATION "the error message output"
::PLURAL.CLASS.TRANSLATION "the error message output"
::BLAND.INSTANCE.TRANSLATION "the error message output"
END.DEFINE

```

```

DEFINE CLASS frequency.standard
::NUMBER.INSTANCES 1
::ANNOUNCEMENT
new.line()!
"A model of the"!
" frequency standard was created."
::CLASS.TRANSLATION "the frequency standard"
::FULL.INSTANCE.TRANSLATION "the frequency standard"
::PLURAL.CLASS.TRANSLATION "the frequency standard"
::BLAND.INSTANCE.TRANSLATION "the frequency standard"
END.DEFINE

DEFINE CLASS gyro.buffer.eca
::NUMBER.INSTANCES 1
::ANNOUNCEMENT
new.line()!
"A model of the gyro buffer ECA"!
" was created."
::CLASS.TRANSLATION "the gyro buffer ECA"
::FULL.INSTANCE.TRANSLATION "the gyro buffer ECA"
::PLURAL.CLASS.TRANSLATION "the gyro buffer ECA"
::BLAND.INSTANCE.TRANSLATION "the gyro buffer ECA"
END.DEFINE

DEFINE CLASS gyroscope
::NUMBER.INSTANCES 1
::ANNOUNCEMENT
new.line()! "A model of the gyroscope"
!" was created."
::CLASS.TRANSLATION "the gyroscope"
::FULL.INSTANCE.TRANSLATION "the gyroscope"
::PLURAL.CLASS.TRANSLATION "the gyroscope"
::BLAND.INSTANCE.TRANSLATION "the gyroscope"
END.DEFINE

DEFINE CLASS gyro.speed.control.function.error
::NUMBER.INSTANCES 1
::CLASS.TRANSLATION "the gyro speed control function"
::FULL.INSTANCE.TRANSLATION "the gyro speed control function"
::PLURAL.CLASS.TRANSLATION "the gyro speed control function"
::BLAND.INSTANCE.TRANSLATION "the gyro speed control function"
END.DEFINE

DEFINE CLASS gyro.speed.control.function.output
::NUMBER.INSTANCES 1
::ANNOUNCEMENT
new.line()! "A model of the gyro speed "!"
"control function was created."
::CLASS.TRANSLATION "the gyro speed control function"
::FULL.INSTANCE.TRANSLATION "the gyro speed control function"
::PLURAL.CLASS.TRANSLATION "the gyro speed control function"
::BLAND.INSTANCE.TRANSLATION "the gyro speed control function"
END.DEFINE

```

```

DEFINE CLASS                                gyro.torque.function
::NUMBER.INSTANCES                          1
::ANNOUNCEMENT                             nev.line()!
                                              "A model of the gyro torque function"!
                                              " was created."
::CLASS.TRANSLATION                        "the gyro torque function"
::FULL.INSTANCE.TRANSLATION                "the gyro torque function"
::PLURAL.CLASS.TRANSLATION                 "the gyro torquing function"
::BLAND.INSTANCE.TRANSLATION               "the gyro torquing function"
END.DEFINE

DEFINE CLASS                                gyro.torquer
::NUMBER.INSTANCES                          1
::ANNOUNCEMENT                             nev.line()! "A model of the gyro " !
                                              "torquer was created."
::CLASS.TRANSLATION                        "the gyro torquer"
::FULL.INSTANCE.TRANSLATION                "the gyro torquer"
::PLURAL.CLASS.TRANSLATION                 "the gyro torquer"
::BLAND.INSTANCE.TRANSLATION               "the gyro torquer"
END.DEFINE

DEFINE CLASS                                gyro.torquer.field.return
::NUMBER.INSTANCES                          1
::ANNOUNCEMENT                             nev.line()!
                                              "A model of the gyro torquer "!
                                              "field return was created."
::CLASS.TRANSLATION                        "gyro field return"
::FULL.INSTANCE.TRANSLATION                "the gyro torquer field return"
::PLURAL.CLASS.TRANSLATION                 "the gyro torquer field return"
::BLAND.INSTANCE.TRANSLATION               "the gyro torquer field return"
END.DEFINE

DEFINE CLASS                                path
::NUMBER.INSTANCES                          any
::CLASS.TRANSLATION                        "the platform torquing function"
::FULL.INSTANCE.TRANSLATION                "a path"
::PLURAL.CLASS.TRANSLATION                 "the platform torquing function"
::BLAND.INSTANCE.TRANSLATION               "the platform torquing function"
END.DEFINE

DEFINE CLASS                                platform.torquing.function
::NUMBER.INSTANCES                          1
::ANNOUNCEMENT                             nev.line()!
                                              "A model of the platform"!
                                              " torquing function was created."
::CLASS.TRANSLATION                        "the platform torquing function"
::FULL.INSTANCE.TRANSLATION                "the platform torquing function"
::PLURAL.CLASS.TRANSLATION                 "the platform torquing function"
::BLAND.INSTANCE.TRANSLATION               "the platform torquing function"
END.DEFINE

```

```

DEFINE CLASS                                power.distribution
::NUMBER.INSTANCES                          1
::ANNOUNCEMENT                             nev.line()!
                                              "A model of the power"!
                                              " distribution was created."
::CLASS.TRANSLATION                         "the power distribution"
::FULL.INSTANCE.TRANSLATION                 "the power distribution"
::PLURAL.CLASS.TRANSLATION                  "the power distribution"
::BLAND.INSTANCE.TRANSLATION                "the power distribution"
END.DEFINE

DEFINE CLASS                                power.supply
::NUMBER.INSTANCES                          1
::ANNOUNCEMENT                             nev.line()! "A model of the"!
                                              " power supply was created."
::CLASS.TRANSLATION                         "the power supply"
::FULL.INSTANCE.TRANSLATION                 "the power supply"
::PLURAL.CLASS.TRANSLATION                  "the power supply"
::BLAND.INSTANCE.TRANSLATION                "the power supply"
END.DEFINE

DEFINE CLASS                                precision.current.network
::NUMBER.INSTANCES                          1
::ANNOUNCEMENT                             nev.line()!
                                              "A model of the precision" !
                                              " current network was created."
::CLASS.TRANSLATION                         "the precision current network"
::FULL.INSTANCE.TRANSLATION                 "the precision current network"
::PLURAL.CLASS.TRANSLATION                  "the precision current network"
::BLAND.INSTANCE.TRANSLATION                "the precision current network"
END.DEFINE

DEFINE CLASS                                search
::NUMBER.INSTANCES                          1
::CLASS.TRANSLATION                         "the search"
::FULL.INSTANCE.TRANSLATION                 "the search"
::PLURAL.CLASS.TRANSLATION                  "the search"
::BLAND.INSTANCE.TRANSLATION                "the search"
END.DEFINE

DEFINE CLASS                                sequence.timing.function.run
::NUMBER.INSTANCES                          1
::CLASS.TRANSLATION                         "the sequence timing function"
::FULL.INSTANCE.TRANSLATION                 "the gyro run and start commands"
                                              !" from the sequence timing function"
::PLURAL.CLASS.TRANSLATION                  "the sequence timing function"
::BLAND.INSTANCE.TRANSLATION                "the sequence timing function"
END.DEFINE

```

```

DEFINE CLASS                                sequence.timing.function.torque
::NUMBER.INSTANCES                          1
::ANNOUNCEMENT                             new.line()!
                                           "A model of the sequence timing"
                                           "! " torque function was created."
::CLASS.TRANSLATION                        "the sequence timing function"
::FULL.INSTANCE.TRANSLATION                "the torque commands from "
                                           !"sequence timing function"
::PLURAL.CLASS.TRANSLATION                 "the sequence timing function"
::BLAND.INSTANCE.TRANSLATION               "the sequence timing function"
END.DEFINE

DEFINE CLASS                                slip.ring
::NUMBER.INSTANCES                          1
::ANNOUNCEMENT                             new.line()! "A model of the"!
                                           " slip ring was created."
::CLASS.TRANSLATION                        "the slip ring"
::FULL.INSTANCE.TRANSLATION                "the slip ring"
::PLURAL.CLASS.TRANSLATION                 "the slip ring"
::BLAND.INSTANCE.TRANSLATION               "the slip ring"
END.DEFINE

DEFINE CLASS                                test.equipment
::NUMBER.INSTANCES                          1
::ANNOUNCEMENT                             new.line()! "A model of the test"!
                                           "equipment was created."
::CLASS.TRANSLATION                        "the test equipment"
::FULL.INSTANCE.TRANSLATION                "the test equipment"
::PLURAL.CLASS.TRANSLATION                 "the test equipment"
::BLAND.INSTANCE.TRANSLATION               "the test equipment"
END.DEFINE

DEFINE CLASS                                torque.driver
::NUMBER.INSTANCES                          1
::ANNOUNCEMENT                             new.line()! "A model of the torque driver"!
                                           " was created."
::CLASS.TRANSLATION                        "the torque driver"
::FULL.INSTANCE.TRANSLATION                "the torque driver"
::PLURAL.CLASS.TRANSLATION                 "the torque driver"
::BLAND.INSTANCE.TRANSLATION               "the torque driver"
END.DEFINE

DEFINE CLASS                                velocity.meter.function
::NUMBER.INSTANCES                          1
::ANNOUNCEMENT                             new.line()! "A model of the velocity meter"
                                           "! "function was created."
::CLASS.TRANSLATION                        "the velocity meter function"
::FULL.INSTANCE.TRANSLATION                "the velocity meter function"
::PLURAL.CLASS.TRANSLATION                 "the velocity meter function"
::BLAND.INSTANCE.TRANSLATION               "the velocity meter function"
END.DEFINE

```

```

DEFINE CLASS                                x.velocity.meter
::NUMBER.INSTANCES                          1
::ANNOUNCEMENT                             new.line()! "A model of "!
                                           "the X velocity meter was created."
                                           "the X velocity meter"
::CLASS.TRANSLATION                        "the X velocity meter"
::FULL.INSTANCE.TRANSLATION                "the X velocity meter"
::PLURAL.CLASS.TRANSLATION                 "the X velocity meters"
::BLAND.INSTANCE.TRANSLATION               "the X velocity meter"
END.DEFINE

DEFINE CLASS                                xy.gyroscope
::NUMBER.INSTANCES                          1
::ANNOUNCEMENT                             new.line()!
                                           "A model of the XY gyroscope"
                                           "! was created."
                                           "the XY gyroscope"
::CLASS.TRANSLATION                        "the XY gyroscope"
::FULL.INSTANCE.TRANSLATION                "the XY gyroscope"
::PLURAL.CLASS.TRANSLATION                 "the XY gyroscope"
::BLAND.INSTANCE.TRANSLATION               "the XY gyroscope"
END.DEFINE

DEFINE CLASS                                y.velocity.meter
::NUMBER.INSTANCES                          1
::ANNOUNCEMENT                             new.line()! "A model of "!
                                           "the Y velocity meter was created."
                                           "the Y velocity meter"
::CLASS.TRANSLATION                        "the Y velocity meter"
::FULL.INSTANCE.TRANSLATION                "the Y velocity meter"
::PLURAL.CLASS.TRANSLATION                 "the Y velocity meters"
::BLAND.INSTANCE.TRANSLATION               "the Y velocity meter"
END.DEFINE

DEFINE CLASS                                yz.gyroscope
::NUMBER.INSTANCES                          1
::ANNOUNCEMENT                             new.line()!"A model of the YZ"!
                                           " gyroscope was created."
                                           "the YZ gyroscope"
::CLASS.TRANSLATION                        "the YZ gyroscope"
::FULL.INSTANCE.TRANSLATION                "the YZ gyroscope"
::PLURAL.CLASS.TRANSLATION                 "the YZ gyroscope"
::BLAND.INSTANCE.TRANSLATION               "the YZ gyroscope"
END.DEFINE

```



```

DEFINE ATTRIBUTE
::DEFINED.ON
::TYPE
::MULTIVALUED
::LEGAL.MEANS
::DETERMINATION.MEANS
::TRANSLATION
END.DEFINE

```

```

END.POINT
COMPONENT:imu.component
boolean
false
{try.rules}
{try.rules}
"the component is an endpoint"

```

```

DEFINE ATTRIBUTE
::DEFINED.ON
::TYPE
::MULTIVALUED
::LEGAL.MEANS
::DETERMINATION.MEANS
::PROMPT

::TRANSLATION
END.DEFINE

```

```

ERROR.SIGNAL
gyro.torque.function
boolean
false
{query.user}
{query.user}
"Check the torque current error signal."
!" Is the signal good?"
"the torque current error signal is good."

```

```

DEFINE ATTRIBUTE
::DEFINED.ON
::TYPE
::MULTIVALUED
::LEGAL.MEANS
::DETERMINATION.MEANS
::PROMPT

::TRANSLATION
END.DEFINE

```

```

GYRO.OUTPUT.OK
gyro.speed.control.function.output
boolean
false
{query.user}
{query.user}
"Check the gyro pickoffs."
!" Are the signals from the gyros ok?"
"the gyro outputs are ok"

```

```

DEFINE ATTRIBUTE
::DEFINED.ON
::TYPE
::MULTIVALUED
::TRANSLATION

```

```

HAS.SUBCOMPONENT
COMPONENT:imu.component
boolean
false
instance.trans(COMPONENT) !
verb(" does ", " does not ") !
" contain subcomponents "

```

```

END.DEFINE

```

```

DEFINE ATTRIBUTE
::DEFINED.ON
::TYPE
::MULTIVALUED
::LEGAL.MEANS
::DETERMINATION.MEANS
::PROMPT
::TRANSLATION
END.DEFINE

```

```

LEVEL.PLATFORM
platform.torquing.function
boolean
false
{query.user}
{query.user}
"Is the platform level?"
"the platform is level"

```

DEFINE ATTRIBUTE	MULTIPLE.INPUT
::DEFINED.ON	COMPONENT:imu.component
::TYPE	boolean
::MULTIVALUED	false
END.DEFINE	
DEFINE ATTRIBUTE	NAME.OF
::DEFINED.ON	COMPONENT:imu.component
::TYPE	text
::MULTIVALUED	false
::TRANSLATION	"the name of a circuit component"
END.DEFINE	
DEFINE ATTRIBUTE	OUTPUT.TEST.OK
::DEFINED.ON	COMPONENT:imu.component
::TYPE	boolean
::MULTIVALUED	false
::LEGAL.MEANS	{try.rules,query.user}
::DETERMINATION.MEANS	{try.rules,query.user}
::PROMPT	"Is the output of the "
	instance.trans(COMPONENT) ! " good?"
::TRANSLATION	instance.trans(COMPONENT) ! verb(" did not ",
	" did ") ! "fail the output test"
END.DEFINE	
DEFINE ATTRIBUTE	RISK
::DEFINED.ON	COMPONENT:imu.component
::TYPE	text
::LEGAL.VALUES	{low,med,high}
::MULTIVALUED	false
::TRANSLATION	"the risk of a circuit component"
END.DEFINE	
DEFINE ATTRIBUTE	RUN.COMMAND.OK
::DEFINED.ON	sequence.timing.function.run
::TYPE	boolean
::MULTIVALUED	false
::LEGAL.MEANS	{query.user}
::DETERMINATION.MEANS	{query.user}
::PROMPT	"Check the gyro run and gyro start commands
	from the NCC. Are the signals good?"
::TRANSLATION	"the start and run signals are good."
END.DEFINE	
DEFINE ATTRIBUTE	SAME.OUTPUT.AS
::DEFINED.ON	COMPONENT1:imu.component,
	COMPONENT2:imu.component
::TYPE	boolean
::MULTIVALUED	false
::TRANSLATION	instance.trans(COMPONENT1) ! verb(" does ",
	does not ") ! "have the same output as " !
	instance.trans(COMPONENT2)
END.DEFINE	

DEFINE ATTRIBUTE

::DEFINED.ON
::TYPE
::LEGAL.VALUES
::MULTIVALUED
::TRANSLATION
END.DEFINE

SEARCH.LEVEL

path
text
{low.level,med.level,high.level}
false
"the level of the search"

DEFINE ATTRIBUTE

::DEFINED.ON
::TYPE
::MULTIVALUED
END.DEFINE

TEST.LOW.RISK.INPUTS

COMPONENT:imu.component
boolean
false

DEFINE ATTRIBUTE

::DEFINED.ON
::TYPE
::MULTIVALUED
END.DEFINE

TEST.MED.RISK.INPUTS

COMPONENT:imu.component
boolean
false

DEFINE ATTRIBUTE

::DEFINED.ON
::TYPE
::MULTIVALUED
::LEGAL.MEANS
::DETERMINATION.MEANS
::PROMPT

::TRANSLATION
END.DEFINE

TORQUE.COMMAND.OK

sequence.timing.function.torque
boolean
false
{query.user}
{query.user}
"Check the torque commands from "
"the NCC. Are the signals good?"
"the start and run signals are good."

```

/*****
/*
/*
/*
/* The order of Rules 201, 202, and 203 ensure that high risk inputs
/* will be tested first, then med risk, then low risk.
*/
*/

DEFINE RULE          RULE.201
::APPLIED.TO        COMPONENT:imu.component
::PREMISE            already.existing(COMPONENT1:imu.component |
                        determined?(connected.to[COMPONENT1,COMPONENT])
                        and connected.to [COMPONENT1,COMPONENT] known
                        and risk[COMPONENT1] is high
                        and output.test.ok[COMPONENT1] thought.not)
::CONCLUSION         bad.input[COMPONENT] =
                        connected.to[COMPONENT1,COMPONENT];
END.DEFINE

DEFINE RULE          RULE.202
::APPLIED.TO        COMPONENT:imu.component
::PREMISE            already.existing(COMPONENT1:imu.component |
                        determined?(connected.to[COMPONENT1,COMPONENT])
                        and connected.to [COMPONENT1,COMPONENT] known
                        and risk[COMPONENT1] is med
                        and output.test.ok[COMPONENT1] thought.not)
::CONCLUSION         bad.input[COMPONENT] =
                        connected.to[COMPONENT1,COMPONENT];
END.DEFINE

DEFINE RULE          RULE.203
::APPLIED.TO        COMPONENT:imu.component
::PREMISE            already.existing(COMPONENT1:imu.component |
                        determined?(connected.to[COMPONENT1,COMPONENT])
                        and connected.to [COMPONENT1,COMPONENT] known
                        and risk[COMPONENT1] is low
                        and output.test.ok[COMPONENT1] thought.not)
::CONCLUSION         bad.input[COMPONENT] =
                        connected.to[COMPONENT1,COMPONENT];
END.DEFINE

/*****
/* Rules 204 - 218 change the prompt to the technician to a more
/* meaningful prompt.
*/
*/

DEFINE RULE          RULE.204
::APPLIED.TO        velocity.meter.function
::PREMISE            already.existing(I:imu|
                        error.message[I] is in {velocity.unreasonable,
                                                    vt.greater.than.2.knots})
::CONCLUSION         output.test.ok[velocity.meter.function]<-1.0>
END.DEFINE

```

```

::CONCLUSION
END.DEFINE

```

```

RULE.205
gyro.speed.control.function.error
already.existing(I:imu|
error.message[I] is in {velocity.unreasonable,
                        vt.greater.than.2.knots}))
output.test.ok[gyro.speed.control.function.error]

```

```

::CONCLUSION
END.DEFINE

```

```
RULE.206  
velocity.meter.function  
already.existing(I:imu|  
error.message[I] is in {xy.speed.control,  
yz.speed.control})  
output.test.ok[velocity.meter.function]
```

```

::CONCLUSION
END.DEFINE

```

```
RULE.207  
gyro.speed.control.function.error  
already.existing(I:imu|  
error.message[I] is xy.speed.control)  
output.test.ok[gyro.speed.control.function.error]<-1.0>
```

```

::CONCLUSION
END.DEFINE

```

```
RULE.208  
gyro.speed.control.function.error  
already.existing(I:imu|  
error.message[I] is yz.speed.control)  
output.test.ok[gyro.speed.control.function.error]<-1.0>
```

```

DEFINE RULE
  ::APPLIED.TO
  ::PREMISE
  ::CONCLUSION
END.DEFINE

```

RULE.209
platform.torquing.function
level.platform[platform.torquing.function]
output.test.ok[platform.torquing.function]

```

DEFINE RULE
  ::APPLIED.TO
  ::PREMISE
  ::CONCLUSION
END.DEFINE

```

```
RULE.210
platform.torquing.function
not level.platform[platform.torquing.function]
output.test.ok[platform.torquing.function]<-1.0>
```

```

DEFINE RULE
  ::APPLIED.TO
  ::PREMISE
  ::CONCLUSION
END.DEFINE

```

```
RULE.211  
gyro.speed.control.function.output  
gyro.output.ok[gyro.speed.control.function.output]  
output.test.ok[gyro.speed.control.function.output]
```

```

DEFINE RULE
  ::APPLIED.TO
  ::PREMISE
  ::CONCLUSION
END.DEFINE

```

```

RULE.212
gyro.speed.control.function.output
  not gyro.output.ok[gyro.speed.control.function.output]
output.test.ok[gyro.speed.control.function.output]<-1.0>

```

```

DEFINE RULE
::APPLIED.TO
::PREMISE
::CONCLUSION
END.DEFINE

```

```

RULE.213
gyro.torque.function
error.signal[gyro.torque.function]
output.test.ok[gyro.torque.function]

```

```

DEFINE RULE
::APPLIED.TO
::PREMISE
::CONCLUSION
END.DEFINE

```

```

RULE.214
gyro.torque.function
not error.signal[gyro.torque.function]
output.test.ok[gyro.torque.function]<-1.0>

```

```

DEFINE RULE
::APPLIED.TO
::PREMISE
::CONCLUSION
END.DEFINE

```

```

RULE.215
sequence.timing.function.run
run.command.ok[sequence.timing.function.run]
output.test.ok[sequence.timing.function.run]

```

```

DEFINE RULE
::APPLIED.TO
::PREMISE
::CONCLUSION
END.DEFINE

```

```

RULE.216
sequence.timing.function.run
not run.command.ok[sequence.timing.function.run]
output.test.ok[sequence.timing.function.run]<-1.0>

```

```

DEFINE RULE
::APPLIED.TO
::PREMISE
::CONCLUSION
END.DEFINE

```

```

RULE.217
sequence.timing.function.torque
torque.command.ok[sequence.timing.function.torque]
output.test.ok[sequence.timing.function.torque]

```

```

DEFINE RULE
::APPLIED.TO
::PREMISE
::CONCLUSION
END.DEFINE

```

```

RULE.218
sequence.timing.function.torque
not torque.command.ok[sequence.timing.function.torque]
output.test.ok[sequence.timing.function.torque]<-1.0>

```

```

/*****
/* Rule 300 and 301 define a critical test component as one that is */
/* either high risk or has multiple inputs. A critical component must */
/* be tested on the first pass. */

```

```

DEFINE RULE
::APPLIED.TO
::PREMISE
::CONCLUSION
END.DEFINE

```

```

RULE.300
COMPONENT:imu.component
risk[component] is high
critical.test[component]

```

```

DEFINE RULE
::APPLIED.TO
::PREMISE
::CONCLUSION
END.DEFINE

```

```

RULE.301
COMPONENT:imu.component
multiple.input[component]
critical.test[component]

```

```

/*****
/* Rule 302 ensures that the first component in the subcomponent is */
/* marked as having a bad output to prevent the system from needlessly */
/* prompting for another test. */

```

```

DEFINE RULE      RULE.302
  ::APPLIED.TO  COMPONENT: imu.component
  ::PREMISE     already.existing(COMPONENT1: imu.component |
                  same.output.as[COMPONENT1, COMPONENT] and
                  not output.test.ok[COMPONENT1])
  ::CONCLUSION  output.test.ok[component] <-1.0>
END.DEFINE

```

Bibliography

1. Anderson, C.K. and P. Duong, "Artificial Intelligence as Applied to Advanced ATE," IEEE Autotestcon '86, pp. 181-186 (1986).
2. Ben-Bassat, Moshe et al., "ATEX - A Diagnostic Expert System Embedded in an ATE System," IEEE Autotestcon '86, pp. 153-157 (1986).
3. Bradbury, William B., "ART-FUL Diagnosis: The Rule-Based Go-Chain," IEEE Autotestcon '86, pp. 37-43 (1986).
4. Cantone, Richard R. et al., "IN-ATE: Fault Diagnosis as Expert System Guided Search," Proceedings of the Air Force Workshop on Artificial Intelligence Applications for Integrated Diagnostics, pp. 298-351, July 1987 (AFWAL-TR-87-1101).
5. Davis, Randall, "Diagnosis Via Causal Reasoning: Paths of Interaction and The Locality Principle," Artificial Intelligence in Maintenance, edited by J. Jeffrey Richardson, pp. 102-122. Park Ridge NJ: Noyes Publications, 1985.
6. Davis, Randall, et al., "Diagnosis Based on Description of Structure and Function," Proceedings of the National Conference on Artificial Intelligence, pp. 137-142 (1982).
7. De Jong, Kenneth, "Expert Systems in the ATE Arena," IEEE Autotestcon '85, pp. 129-132 (1985).
8. Department of the Navy, Organizational Level Technical Manual For Inertial Navigation System AN/WSN-1(V)2. NAVSHIPS 0967-529-6010. November 1973.
9. Estes, Authur, L, Knowledge-Based Automatic Test Program Generator for the Abbreviated Test Language For All Systems, MSCS thesis, AFIT/GCS/ENG/86D-18. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1986 (AD-A177-604).
10. Fink, Pamela K. and John C. Lusth, "The Integrated Diagnostic Model - Towards a Second Generation Diagnostic Expert System," Proceedings of the Air Force Workshop on Artificial Intelligence Applications for Integrated Diagnostics, pp. 188-197, July 1987 (AFWAL-TR-87-1101).
11. Harmon, Paul and David King, Expert Systems: Artificial Intelligence in Business. New York: J. Wiley, 1985.
12. Johnson, Tim, The Commercial Application of Expert System Technology. London, Ovum Ltd, 1984.

13. Kunert, Jerry L, "Knowledge Engineering" IEEE Autotestcon '86, pp. 159-164 (1986).
14. Laffey, T.J., et al, "LES: A Model-Based Expert System for Electronic Maintenance," Artificial Intelligence in Maintenance, edited by J. Jeffrey Richardson pp. 414-435. Park Ridge NJ: Noyes Publications, 1985.
15. Liguruori, Fred, Automatic Test Equipment: Hardware, Software, and Management. New York: IEEE Press, 1974.
16. Papenhausen, David W. and Brent L. Hadley, CEPS - An Artificial Intelligence Approach to Avionics Maintenance," Proceedings of the Air Force Workshop on Artificial Intelligence Applications for Integrated Diagnostics, pp. 208-217, July 1987 (AFVAL-TR-87-1101).
17. Pipitone, Frank, "The FIS Electronics Troubleshooting System" Computer, 19, pp. 68-76, (July, 1986).
18. Porter, Kenneth A, "AI Applications to Automatic Testing" IEEE Autotestcon '87, pp. 377-382 (1987).
19. Ramsey, James E. Jr, Diagnosis: Using Automatic Test Equipment and an Artificial Intelligence Expert System, MSEE thesis, AFIT/GE/ENG/84D-53. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1984 (AD-A151-918).
20. Rasmussen, Steven J., "Expert System for Depot Maintenance of the Dual Miniature Inertial Navigation System," 1988 IEEE National Aerospace and Electronics Conference, pp. 1369-1374, May 1988.
21. Rasmussen, Steven J., System Engineer for DMINS Testing. Personal interviews. AGMC, Newark AFB OH, April - September 1988.
22. Schutzer, Daniel, Artificial Intelligence An Applications-oriented Approach. New York: Van Nostrand Rheinhold Company Inc, 1987.
23. Teknowledge, Inc., S.1 User's Guide, Palo Alto, CA 1986.
24. Waterman, Donald, A Guide to Expert Systems. Reading MA: Addison-Wesley Publishing Company, 1986.
25. Wunz, Donald R. Jr, Diagnosis of Analog Electronic Circuits: A Functional Approach, MSCE thesis, AFIT/GCS/ENG/86M-3. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, March 1986 (AD-A172-718).
26. Yazdani, Masoud, Artificial Intelligence Principles and Applications. London: Chapman and Hall Ltd, 1986.

VITA

James M. Skinner [REDACTED]

[REDACTED] 1972 [REDACTED] first enlisted in the Air Force in May 1974. After serving six years he separated from the service and attended college at Miami Dade Community College, where in 1981, he received an Associates Degree with Highest Honors. In 1984, he received a Bachelor of Science in Electrical Engineering from the University of Washington. That same year, he was commissioned in the USAF and was assigned to the Avionics Laboratory where he served as the contract manager of the High Accuracy Ring Laser Gyro from 1985 to 1987. In 1987 he was selected to enter the Air Force Institute of Technology for an advanced degree in Computer Engineering, specializing in artificial intelligence.

[REDACTED]

[REDACTED]

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS	
2a. SECURITY CLASSIFICATION AUTHORITY			3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.	
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE			5. MONITORING ORGANIZATION REPORT NUMBER(S)	
PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCB/ENG/88D-5				
4a. NAME OF PERFORMING ORGANIZATION School of Engineering		4b. OFFICE SYMBOL (If applicable) AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION	
5c. ADDRESS (City, State, and ZIP Code) Air Force Institute of Technology Wright-Patterson AFB OH 45433-6583			7b. ADDRESS (City, State, and ZIP Code)	
6a. NAME OF FUNDING / SPONSORING ORGANIZATION Inertial Engineering		6b. OFFICE SYMBOL (If applicable) AGHC/MAEEK	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
7c. ADDRESS (City, State, and ZIP Code) Aerospace Guidance and Metrology Center Navark AFB OH 43057-5000			10. SOURCE OF FUNDING NUMBERS	
			PROGRAM ELEMENT NO.	PROJECT NO.
			TASK NO.	WORK UNIT ACCESSION NO.
1. TITLE (Include Security Classification) A Diagnostic System Blending Deep and Shallow Reasoning (U)				
2. PERSONAL AUTHOR(S) James H. Skinner, B.S., Capt, USAF				
13a. TYPE OF REPORT MSCE Thesis		13b. TIME COVERED FROM _____ TO _____		14. DATE OF REPORT (Year, Month, Day) 1988 December
				15. PAGE COUNT 188
6. SUPPLEMENTARY NOTATION				
7. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)	
FIELD	GROUP	SUB-GROUP	Expert Systems, Deep Reasoning, Shallow Reasoning, Diagnostics, Automatic Test Equipment	
12	05			
12	09			
9. ABSTRACT (Continue on reverse if necessary and identify by block number)				
Thesis Chairman: Professor F. M. Brown				
Most expert systems employed with automatic test equipment (ATE) belong to one of two broad categories: shallow reasoning systems, which encode empirical knowledge into a set of IF-THEN rules; and deep reasoning systems, which reason from an understanding of structure and function. Shallow reasoning is useful for incorporating diagnostic experience but is unable to accommodate situations that have not been previously encountered. Deep reasoning is suitable for a new system, but fails to take advantage of the technician's experience when applied to older systems. (Cont'd on back)				
20. DISTRIBUTION / AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL James H. Skinner, Captain, USAF			22b. TELEPHONE (Include Area Code) (513) 255-3030	22c. OFFICE SYMBOL AFIT/ENG

FD Form 1473, JUN 86

Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

UNCLASSIFIED

This thesis describes BDS (Blended Diagnostic System), an AT-based expert system which emulates a human technician by combining shallow and deep reasoning. BDS first applies shallow techniques derived from the technician's experience. If this does not locate the fault, BDS constructs a model of the UUT and guides the technician from place to place in the model, using heuristics based on the likelihood of a component's being faulty and the time required to test the component. This blending of shallow and heuristically-guided deep reasoning extends the class of diagnosable faults; it also reduces the number and duration of tests.

A prototype of BDS was developed for the Dual Miniature Inertial Navigation System and has been delivered to Newark AFB, OH for evaluation.

UNCLASSIFIED